

Group 14

Ahash Mathivathan
ahmat18@student.sdu.dk

Frederik Haagensen
frhaa18@student.sdu.dk

Kasper Delaxson Sebastiampillai
kaseb18@student.sdu.dk

Github:
https://github.com/KasperDelaxson/EmbeddeLinux_Team14

Solution Approach

To build a plant watering system, the architecture design will be the deciding factor in realizing the non-functional requirements. [13]

To develop the system, the UNIX philosophy of “Build a prototype as soon as possible” will be the main driving factor. As such, the focus will be a working version of the design. From a working prototype, different responsibilities will be identified to develop a system that encourages small scripts with a specialized responsibility that lives up to the main tenets of UNIX philosophy. [15]

Architecture

The overall system architecture consists of three devices, namely a Raspberry Pi (RPi) [14], Raspberry Pi Pico [21], and an ESP8266 [11]. The communication between the RPi and Raspberry Pi Pico occurs via the serial port interface, while the ESP8266 utilizes an application programming interface (API) to expose endpoints for data retrieval.

Subsequently, the RPi employs a Message Queue Telemetry Transport (MQTT) broker to manage the data received from the devices. [2] The data is published to the MQTT broker with distinct topics associated with the transmitted sensor data. A Telegraf agent is then employed to read the data from the MQTT broker and transfer it to InfluxDB. Grafana retrieves the data from the InfluxDB and employs it for the purpose of data visualization. Finally, utilization of a PHP-based server encompassing a frontend interface designed for the purpose of health monitoring functionalities pertaining to both the system and internet. The RPi is configured as a WiFi access point, in accordance with the Functional requirement L (F:L), which mandates the ability of the RPi to operate independently of internet connectivity. By encapsulating all functionalities within the RPi itself, the technologies employed maintain their operational capabilities even in the absence of network connectivity.

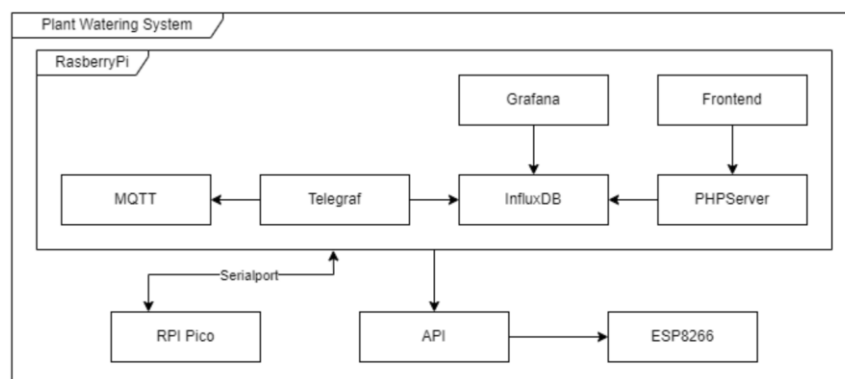


Figure 1 - Architecture of Plant Watering System

MQTT

The non-functional requirements are divided into constraints and qualities which the system needs to express. The constraints are limited to Non-Functional Requirement B (NF:B), and NF:C. which states that the functionality will be built primarily in scripts, and use MQTT as communication.

MQTT is a publish subscribe pattern which realizes a decoupled and event-based system. MQTT consists of a broker, which keeps track of publishers and subscribers as an intermediary, by using the concept of topics. Topics specify data streams to publish/subscribe to and are simple strings with the possibility of wildcards. MQTT topics consist of levels in a directory type style. Levels are differentiated with “/” between them. The “+” wildcard will accept all inputs at that specific level, and therefore subscribe to multiple topic streams. Each client will receive or send messages without knowing who or how many will receive the messages.[16]

By utilizing the MQTT broker the system will express scalable properties with a proper design of topics.[16] Additionally, the decoupling of clients means different shell scripts can perform different tasks with the same data. Each MQTT client will be able to be concise and perform a single task well. Furthermore, multiple instances of the same functionality can be utilized in future developments to distribute and increase performance.[16]

A Quality of Service (QoS) of 2 will be used which guarantees messages will be read or published exactly once at the cost of more latency.[16] Introducing latency can pose an issue regarding fast response times to alarms but will be mitigated by setting alarm states before initializing other event streams.

As a result, MQTT will be a suitable technology to facilitate communication between decoupled scripts in a scalable manner with the main tenets of UNIX in mind.[15]

To utilize MQTT on the RPi without requiring an internet connection, a Mosquitto broker will be used. Mosquitto is a lightweight broker without horizontal scaling.[5] The broker will be able to handle multiple clients at once before suffering performance loss. However, larger deployments of the plant watering system would require a scalable implementation of an MQTT broker.[19]

To accommodate the challenge of maintaining state for each plant in the system, plain ASCII files will be used, which satisfy the philosophy of “Store numerical data in flat ASCII files”.[15] The files will contain the latest state to avoid a steady expansion which can impact storage space. The files will therefore not be used for historic logging of data but as current state holders. Each file will remain human readable and be easy to manage without any specialized tool.[22]

Direct usage of files is not horizontally scalable. To allow for management of multiple plants in a network of RPis, a distributed file system would alleviate the scalability problems while still using text files.[24]

InfluxDB

InfluxDB, developed by InfluxData, is a Time Series Database (TSDB) designed to efficiently store data based on time and value as key-value pairs. To facilitate data management, InfluxDB uses Structured Query Language (SQL) for handling queries.[20]

InfluxDB enables storage of data, associating each entry with a timestamp and a corresponding field set containing the actual data. This proves particularly valuable for logging purposes and maintaining a comprehensive record of sensor data. Consequently, InfluxDB will serve as the repository for all sensor data within the system, thereby facilitating subsequent data visualization. Furthermore, InfluxDB will also assume the responsibility of storing historical performance and health logs, ensuring comprehensive monitoring of the system and its internet connectivity.[23]

Telegraf

Telegraf, developed by InfluxData, is an open-source server agent designed to facilitate collection and transmission of data across various components within a system. Acting as an intermediary entity, Telegraf assumes the role of transferring data between data sources. Notably, Telegraf has a compact memory footprint, rendering it an ideal choice as a data agent, particularly for resource constrained IoT devices. Furthermore, Telegraf is a lightweight service, requiring minimal code modifications to facilitate its utilization. Configuration adjustments primarily involve modifying various plugins, including inputs, processors/aggregators, and outputs in the config file.

Telegraf serves as the conduit for transferring data from a MQTT Broker to an InfluxDB. The input plugin of MQTT, allows for the flexibility of utilizing topic Wild Cards.[8]

Grafana

Grafana is a visualization tool that operates as a dashboard platform, proficient in presenting data in a visually intuitive manner. Its fundamental functionality revolves around retrieving data from different data sources and leveraging it to generate graphs and visual representations.

Grafana serves as a web interface, effectively showcasing monitoring logs from the RPi. By integrating with InfluxDB, Grafana enables the visualization of historical data.[9]

Solution Description

The following section will describe the implementation of some of the key concepts regarding the design and requirements. See Figure 2, for an internal design and setup of MQTT topics and scripts within the RPi.

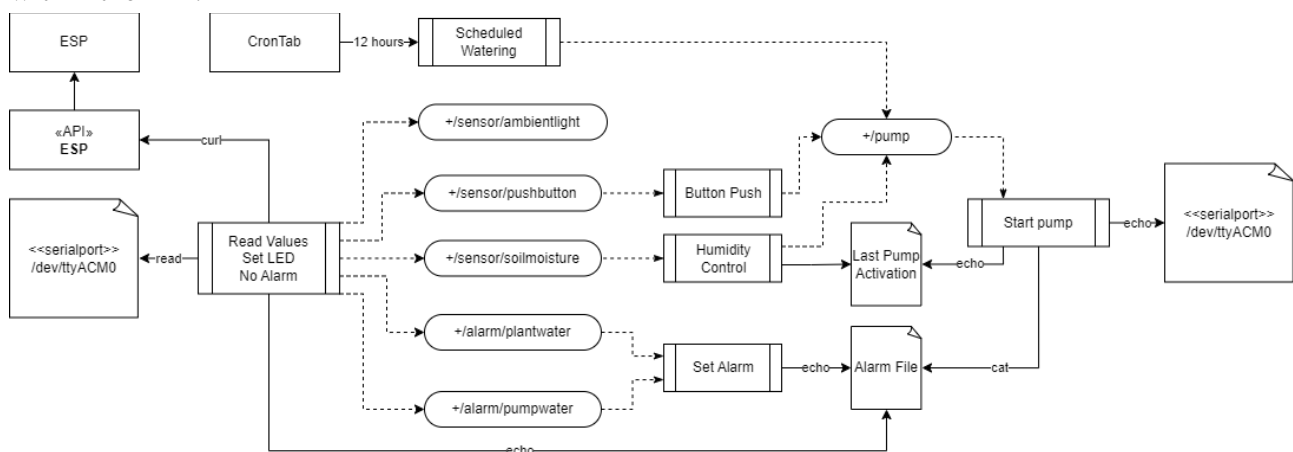


Figure 2 - Script and MQTT Topic Design in the RPi

MQTT

To facilitate the use of MQTT within the project, they are developed as very small scripts. These lay the foundation of starting and ending event processes and shape the system through the design of topic strings. The topics are designed for a plant as “id/type/stream”. “Id” is the unique identifier for each plant. In this prototype a single plant is used through serial ports, which are used as the id.

“Type” is a descriptive name indicating the type of stream to connect to. Therefore, to handle multiple plants, the wildcard “+” is used at the id level. This allows for integration of new plants without any changes to the system infrastructure.

The script to read is a single line based on “mosquitto_sub”.[6] It sets some specific values which will always be present within the plant system, which is username, password and QoS. The only parameters to set are the topic, and an extra parameter which is used for the “-v” tag, to receive the message as “topic payload” for identifying a plant id. The script echoes its results and can be used in the beginning of a pipeline or within a script as needed.

To write to MQTT, the same ideas are applied as in the write script with “mosquitto_pub”.[7] The only difference is the fact that the second parameter accepts the payload. Additionally, it will end a pipeline to finalize an event process and send the data back into MQTT.

Maintaining State of Plant:

Two states are maintained to realize the requirements. Separate files are used for keeping track of the last timestamp of the pump running, and to indicate an alarm or safe state.

Files regarding each state will be kept in separately managed folders wherein each plant will have their individual file. In Figure 4, utilization of the alarm file is shown. All files will be overwritten when new data is retrieved to make only the last messages read. This results in small and separate files, which are easy to use in other scripts. In addition, separating state of each plant minimizes concurrent read or write to the same file.

To maintain the alarm state, an integer of 0 or 1 is used to indicate the current state. The alarm state will be used in the “start pump” script to enforce that the pump will not start in case of any alarms. To enforce that soil moisture starts the pump maximum once every hour, a state file will keep track of the latest activation of the pump as a UNIX timestamp.[1]

InfluxDB

In InfluxDB, a database named “plant_watering_system” has been created, serving as the primary repository for data storage. To organize and manage the incoming sensor and alarm data, a measurement called “AlarmsAndSensors” has been defined. Furthermore, to manage the logging data of the system health the measurement “HealthLogging” has been defined. The implementation of InfluxDB currently operates locally on the RPi device. This configuration ensures the uninterrupted functionality of InfluxDB, even in scenarios where WiFi connectivity is unavailable. By adhering to the requirement F:N, the system effectively fulfils the task of storing performance and health data concerning both the system and internet connectivity. These metrics are persistently logged and stored within the InfluxDB database, facilitating monitoring of the system.

Telegraf

Telegraf input and output is set through the config file. The input is set to the MQTT broker, listening on topics with implemented wildcards. The output is set to InfluxDB, storing all data to the

database "plant_watering_system", and publishing to "AlarmsAndSensors" & "HealthLogging" measurements respectively. Furthermore, Telegraf provides the capability to set the QoS level, which in the current setup is QoS 2.[8]

```
# # Configuration InfluxDB 2.0
[[outputs.influxdb]]
  urls = ["http://127.0.0.1:8086"]
  database = "plant_watering_system"
  username = "telegraf"
  password = "emli"

# # Read metrics from MQTT
[[inputs.mqtt_consumer]]
  servers = ["tcp://localhost:1883"]
  client_id = "ConsumerInteger"
  username = "my_user"
  password = "my_password"
  data_format = "value"
  data_type = "integer"
  name_override = "AlarmsAndSensors"
  topics = [
    "+/alarm/plantwater",
    "+/alarm/pumpwater",
    "+/sensor/soilmoisture",
    "+/sensor/ambientlight",
    "+/sensor/pushbutton",
    "+/pump/activation"
  ]

# # Read metrics from MQTT cont.
[[inputs.mqtt_consumer]]
  servers = ["tcp://localhost:1883"]
  client_id = "ConsumerLoggin"
  username = "my_user"
  password = "my_password"
  data_format = "value"
  data_type = "string"
  name_override = "HealthLogging"
  topics = [
    "logging/+"
  ]

# # QoS
qos = 2
```

Figure 3 - Telegraf Configuration

Grafana

The requirements F:K and F:N are addressed by the system's ability to provide visual representations of historical sensor data. The data is presented in the form of graphs, affording users a comprehensive and intuitive understanding of the collected information. Moreover, the system incorporates functionality that enables users to download the underlying data associated with each graph. This is covered in the accompanying video.

PHP-Server

The monitoring of system health and performance were facilitated through the utilization of an Apache2 web server, which served as the hosting platform for a dedicated web page.[10] This web page was designed to provide insights into the system's health and internet connectivity. Data was retrieved from the "HealthLogging" measurements within the "plant_watering_system" InfluxDB. The implemented solution features a webpage, presenting a table of the most recent monitoring information. These metrics included disk space usage, available RAM, CPU load, CPU temperature, and the volume of bytes transferred across the network. The collection of health logs occurs automatically every 15th minute, through the Crontab.[18] The outcomes of this, along with the user interface, are presented in the video.

Security

The requirement NF:E, which focuses on enhancing the security of the RPi against malicious cyber-attacks, has been addressed to a certain extent in terms of best practices for security.[17]

The first security aspect addresses the prevention of brute force login attempts via SSH and other protocols. Fail2Ban, a monitoring tool, has been employed to track login attempts and automatically ban IP addresses that exceed a predefined threshold. In the RPi, three failed login attempts result in an IP-ban of ten minutes. The monitoring window, referred to as the "findtime," is also set to ten minutes, defining the time frame during which fail2ban monitors login activity.[3]

The second security measure involves the use of SSH keys for authentication instead of traditional username and password combinations. On the Linux desktop, a private and public key pair is generated, and the public key is subsequently transferred to the RPi. The key is stored in a hidden folder named "~/.ssh", which is not visible when using the "ls" command to list directory contents. Furthermore, the permissions on the ".ssh" folder are set to 700, ensuring that only the directory owner has read, write, and execute permissions. [4]

The third security enhancement entails the use of "nftables" instead of "iptables". Nftables provides compatibility with IPv6, making it a more comprehensive and future-proof solution.[12]

Write to Pump

The pump is controlled using the Serialport. The Raspberry Pi Pico is programmed to listen for the character "p" received via the Serialport. Upon receiving this character, the pump is activated and runs for a duration of 1 second. Consequently, the system fulfils requirements F:D, F:E and F:F, which pertain to the ability to control the pump.

All three requirements are satisfied differently, by adhering to their respective constraints.

However, all three will publish to the same MQTT topic stream which is described as "+/pump". A single pipeline will have the responsibility of listening for pump requests and starting the pump if no alarm is present (See Figure 4). If the pump is activated, the resulting timestamp of starting the pump will be published to a topic and written to a log file.

```
#!/bin/bash
bash ./sub_mqtt_topic.sh +/pump -v | while read -r message; do
    IFS=" " read -ra topic_payload <<< "$message"

    IFS="/" read -ra split_topic <<< "${topic_payload[0]}"

    humidity=${topic_payload[1]}
    id=${split_topic[0]}
    file=./alarm/status/${id}.txt
    status=$(cat "${file}")

    if [ "$status" = "0" ]; then
        echo p > /dev/ttyACM0
        current_timestamp=$(date +%s)
        file=./humidity_control_pump/water_timestamps/last_water_${id}.txt
        bash publish_mqtt_message.sh "${id}/pump/activation" "$current_timestamp"
        echo "$current_timestamp" > "${file}"
    fi
done
```

Figure 4 - start_pump.sh responsible for activating the pump

To provide a plant with water every 12 hours, a cronjob is used which executes a simple script to publish a pump request to MQTT. The cron job setting is "0 */12 * * *" to facilitate a pump request at 12:00 and 24:00 each day.[18]

```
#!/bin/bash
bash /home/pi/bin/assignment/publish_mqtt_message.sh e6600883070c612c0241D33C/pump p
```

Figure 5 - "scheduled_watering.sh" run in a 12 hour interval

To satisfy requirement F:E, a pipeline will constantly monitor the soil moisture topic streams. The pipeline will use the topic to find the id of the plant. The pipeline will compare the soil moisture to a cut off value. If the soil moisture is below 20, the pipeline will compare the current timestamp to the last pump activation stored. If more than an hour has passed, the script will publish a pump request.

```

file=./humidity_control_pump/water_timestamps/last_water_${id}.txt
if [ -f "${file}" ]; then
    last_water=$(cat "${file}")
else
    last_water=0
fi

current_timestamp=$(date +%s)
time_difference=$((current_timestamp - last_water))

if [ "$time_difference" -gt 3600 ]; then
    echo "publish"
    bash publish_mqtt_message.sh "${id}/pump" p
fi
done

```

Figure 6 - "last_watering.sh" which listens and requests pump activation maximum once every hour

To satisfy the requirement F:F, the system incorporates a push button functionality that triggers the pump. According to the requirement, if the button is pressed once within a two-second timeframe, the pump should activate. The system achieves this by utilizing an API that provides the number of buttons pushes as a response. After each API request, the push button count is reset, and hence requires reading twice from the API in order to ascertain its compliance with the requirement. The video demonstration showcases incorrect controls, wherein the button must be pressed twice. However, this issue has since been fixed, ensuring that the requirement is now being correctly met.

```

#!/bin/bash
serialnumber=$(cat serialport/SerialNumber.txt)
while true; do
    initial_count=$(curl -s http://192.168.10.222/button/a/count)

    if [[ $initial_count -eq 1 ]]; then
        sleep 2
        updated_count=$(curl -s http://192.168.10.222/button/a/count)
        if [[ $updated_count -eq 0 ]]; then
            bash publish_mqtt_message.sh "$serialnumber/pump" "p"
        fi
    fi
done

```

Figure 7 - "push_button.sh" listening for button pushes and requests pump activations.

Boot

To reinitialize all necessary script pipelines after system shutdown, a bootup script is used. The script contains all functionality for the plant watering system and starts them when the RPi is turned on. Since all the functionalities would be blocking the rest of the scripts, they are all run in the background with the keyword "&".

```

#!/bin/bash
cd "$(dirname "$0")"

stty -F /dev/ttyACM0 115200 -ixon -ixoff
bash ./serialport/serialport_publish_mqtt.sh &
bash ./humidity_listen_and_start_pump.sh &
bash ./start_pump.sh &
bash ./alarm/cancel_all.sh &
bash ./serialport/push_button.sh &

```

Figure 8 - boot up script that runs on every system startup. Starts scripts in background process.

Set LED

To realize the requirements of setting the LEDs on the ESP, all the values from the serial port are necessary. As a result, setting the LEDs is incorporated into the process of reading serial port values and publishing them to MQTT. The functionality is set up to accept one LED colour, which is set to be turned on, while the remaining LEDs will be turned off (See Figure 9).

```
command=$1
states=("green" "yellow" "red")
for state in "${states[@]}"
do
    if [ "$command" = "$state" ]; then
        set_led "$state" "on"
        continue
    fi
    set_led "$state" "off"
done
```

Figure 9 - script for settings the LED, while allowing for at max one LED state to on.

Set Alarm

To stop the pump from running, the alarm state is set by listening to all alarm topic streams (See Figure 10). Whenever an alarm is activated, a value of 1 will subsequently be read by the Serialport and published to its MQTT topic. Whenever a “1” is received by the “cancel all” script, the file containing the current alarm state will be set, which results in blocking any pump activations.

```
#!/bin/bash
bash ./sub_mqtt_topic.sh +/alarm/+ -v | while read -r message; do
    IFS=" " read -ra topic_payload <<< "$message"

    IFS="/" read -ra split_topic <<< "${topic_payload[0]}"
    alarm=${topic_payload[1]}
    id=${split_topic[0]}

    if [ "$alarm" = "1" ]; then
        file=./status/${id}.txt
        echo "$alarm" > "$file"
    fi
done
```

Figure 10 - “cancel_all.sh” to stop pump activations if an alarm is registered

Tests and Results

Validation testing was done to ensure correctness of implementation to fulfil the requirements. Testing was performed by systematically starting scripts and activating event processes. Validation testing was performed on watering every 12 hours, watering based on humidity, validating the pump only runs when no alarm is activated, and that the alarm is set correctly when values are measured.

Start Pump

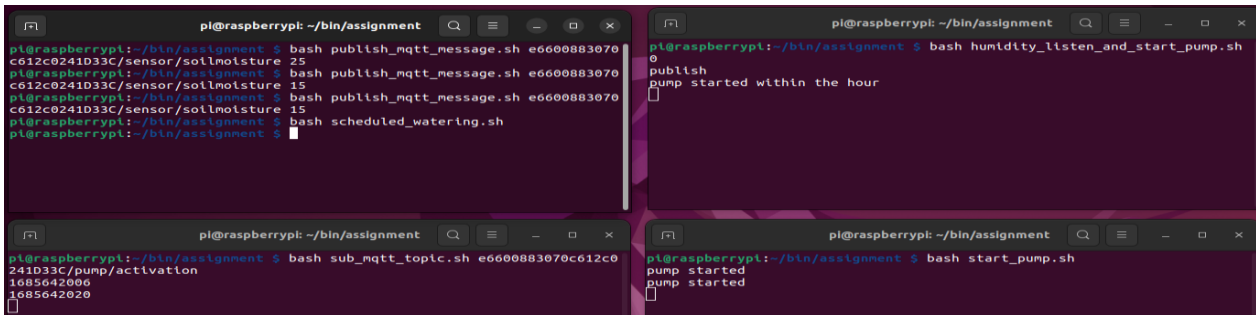
The following section documents how requirements F:D and F:E were tested.

First the script “humidity_listen_and_start_pump” was started. Secondly, to verify the output, the script “start_pump” was started. In Figure 11 the results can be seen. The top left terminal shows initialization of event processes. The bottom left script listens on the topic stream with timestamps of when the pump was started. Four event processes were started sequentially. The first sends in a soil moisture value above the limit. The second and third initialization sends values below the limit. The last script “scheduled_watering” is the script which would run every 12 hours. This is run to test whether this is affected by the humidity event process.

The expected result was:

1. Do not start pump
2. Start pump
3. Do not start pump because of timing constraint.
4. Start pump

The results showcase that the first event returns 0, and therefore does not start the pump. Second event started the pump. The third did not, as seen by the text “pump started within the hour”. The last event makes the pump run again, showing that only the humidity reader is affected by the hourly constraint.



```
pi@raspberrypi: ~/bin/assignment
pi@raspberrypi:~/bin/assignment $ bash publish_mqtt_message.sh e6600883070c612c0241033c/sensor/sollmoisture 25
pi@raspberrypi:~/bin/assignment $ bash publish_mqtt_message.sh e6600883070c612c0241033c/sensor/sollmoisture 15
pi@raspberrypi:~/bin/assignment $ bash publish_mqtt_message.sh e6600883070c612c0241033c/sensor/sollmoisture 15
pi@raspberrypi:~/bin/assignment $ bash scheduled_watering.sh
pi@raspberrypi:~/bin/assignment $

pi@raspberrypi:~/bin/assignment $ bash humidity_listen_and_start_pump.sh
0
publish
pump started within the hour
[]

pi@raspberrypi:~/bin/assignment $ bash sub_mqtt_topic.sh e6600883070c612c0241033c/pump/activation
1685642006
1685642020
[]

pi@raspberrypi:~/bin/assignment $ bash start_pump.sh
pump started
pump started
[]
```

Figure 11 - Test regarding activation of the pump

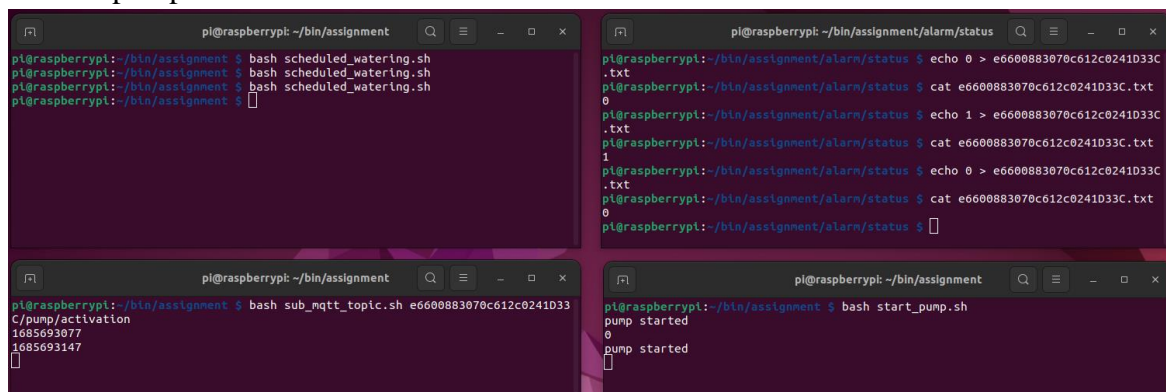
Alarm

To test the alarm, the file used for indicating the state of the alarm were set to either 0 or 1. The script managing the pump and the script to listen on the MQTT topic stream with timestamps of pump activation were started.

In Figure 12 the test can be seen, including input and output. The top right terminal shows insertion of alarm states manually. The test will consist of three steps. The process consists of first setting the plant to a safe state, then an alarm state, and finally back to a safe state.

To test the outcome, the script handling pump interaction is started to validate the outcomes, while the script “scheduled_watering” will be run as it is not affected by time constraints. The expected results were:

1. Start pump
2. Do not start pump because of alarm state.
3. Start pump



```
pi@raspberrypi:~/bin/assignment
pi@raspberrypi:~/bin/assignment $ bash scheduled_watering.sh
pi@raspberrypi:~/bin/assignment $ bash scheduled_watering.sh
pi@raspberrypi:~/bin/assignment $

pi@raspberrypi:~/bin/assignment/alarm/status
pi@raspberrypi:~/bin/assignment/alarm/status $ echo 0 > e6600883070c612c0241033c/.txt
pi@raspberrypi:~/bin/assignment/alarm/status $ cat e6600883070c612c0241033c/.txt
0
pi@raspberrypi:~/bin/assignment/alarm/status $ echo 1 > e6600883070c612c0241033c/.txt
pi@raspberrypi:~/bin/assignment/alarm/status $ cat e6600883070c612c0241033c/.txt
1
pi@raspberrypi:~/bin/assignment/alarm/status $ echo 0 > e6600883070c612c0241033c/.txt
pi@raspberrypi:~/bin/assignment/alarm/status $ cat e6600883070c612c0241033c/.txt
0
pi@raspberrypi:~/bin/assignment/alarm/status $

pi@raspberrypi:~/bin/assignment
pi@raspberrypi:~/bin/assignment $ bash sub_mqtt_topic.sh e6600883070c612c0241033c/pump/activation
1685693077
1685693147
[]

pi@raspberrypi:~/bin/assignment $ bash start_pump.sh
pump started
0
pump started
[]
```

Figure 12 - Test of reliance of using the alarm file

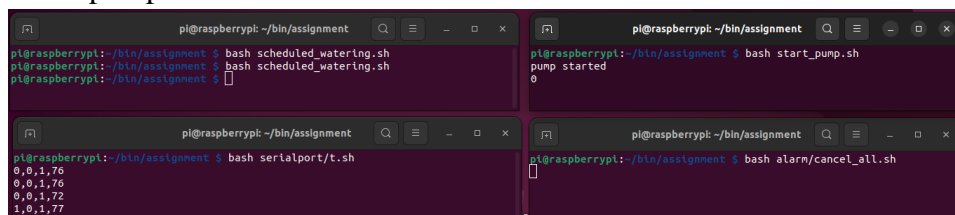
The results as seen in Figure 12 showcases that the script handling pump interaction will not run if an alarm is triggered.

Setting Alarm State

To validate that the alarm state is set according to the data retrieved from the serialport, the following test will start scripts responsible for managing the alarm state as well as the “start_pump” script. First, it will be attempted to activate the pump while no alarm is initiated. Afterwards an alarm will be started by connecting an alarm wire pair. After the alarm is started, the script to start the pump will run again.

The expected results are:

1. Start pump
2. Do not start pump



The figure displays four terminal windows from a Raspberry Pi. The top-left window shows the execution of 'scheduled_watering.sh' twice. The top-right window shows 'start_pump.sh' being executed, resulting in 'pump started' and a '0' return code. The bottom-left window shows 'serialport/t.sh' being executed, displaying four rows of sensor data: '0,0,1,76', '0,0,1,76', '0,0,1,72', and '1,0,1,77'. The bottom-right window shows 'alarm/cancel_all.sh' being executed.

Figure 13 - Setting alarm state test

The results of the initiation of scripts can be seen in Figure 13. The achieved and expected results align in their outcome.

Conclusion

The report presents the project work carried out for an embedded plant watering system, with the main tenets of UNIX philosophy in mind. The system incorporates various hardware components, including the RPi, Raspberry Pi Pico, and ESP8266. The system has been designed to emphasize scalability, based on the non-functional requirements. To facilitate decoupled and scalable communication, the system employs the MQTT lightweight communication protocol. Additionally, a Telegraf agent has been utilized for data transfer. For the purpose of data storage and management, an InfluxDB database has been employed, allowing the storage of sensor, alarm, and logging data as time series. To visualize the collected sensor and alarm data, a Grafana dashboard has been developed. Furthermore, a PHP-based server with a web interface has been implemented to monitor the system's health. Security measures, including fail2ban, ssh-key authentication, and nftables, have been incorporated to enhance the system's security and reliability. To ensure system availability, a bootup script has been created to automatically start the system after a shutdown. The report documents the fulfilment of functional requirements, primarily through shell scripts. Validation testing has been conducted to verify the correct implementation of the requirements. Finally, a video demonstration has been created to showcase the operational functionality of the system.

REFERENCES

- [1] [n. d.]. <https://www.unixtimestamp.com/>
- [2] [n. d.]. <https://mqtt.org/>
- [3] [n. d.]. https://www.fail2ban.org/wiki/index.php/Main_Page
- [4] [n. d.]. <https://chmodcommand.com/chmod-700/>
- [5] 2018. <https://mosquitto.org/>
- [6] 2021. https://mosquitto.org/man/mosquitto_sub-1.html
- [7] 2021. https://mosquitto.org/man/mosquitto_pub-1.html
- [8] 2023. <https://www.influxdata.com/time-series-platform/telegraf/>
- [9] 2023. <https://grafana.com/>
- [10] 2023. <https://ubuntu.com/server/docs/web-servers-apache>
- [11] 2023. <https://www.espressif.com/en/products/socs/esp8266>
- [12] 2023. https://wiki.nftables.org/wiki-nftables/index.php/Main_Page
- [13] Len Bass, Paul Clements, and Rick Kazman. 2003. Software architecture in practice. Addison-Wesley Professional.
- [14] Raspberry Pi Foundation. 2023. Teach, learn, and make with the Raspberry Pi Foundation. <https://www.raspberrypi.org/>
- [15] Mike Gancarz. 1995. The unix philosophy. Digital Press.
- [16] HiveMQ. 2020. MQTT and MQTT 5 Essentials. <https://www.hivemq.com/mqtt-essentials/>
- [17] Kjeld Jensen. 2023. Course notes, module 6 week 10.
- [18] Michael Kerrisk. 2022. Crontab(5). <https://man7.org/linux/man-pages/man5/crontab.5.html>
- [19] Biswajeeban Mishra, Biswaranjan Mishra, and Attila Kertesz. 2021. Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements. *Energies* 14, 18 (2021). <https://doi.org/10.3390/en14185817>
- [20] Juan C Olivares-Rojas, Enrique Reyes-Archundia, José A Gutiérrez-Gnecchi, Ismael Molina-Moreno, Jaime Cerda-Jacobo, and Arturo Méndez-Patiño. 2021. A Comparative Assessment of Embedded Databases for Smart Metering Systems. In 2021 IEEE PES Innovative Smart Grid Technologies Conference-Latin America (ISGT Latin America). IEEE, 1–5.
- [21] Raspberry Pi. 2023. Buy A raspberry pi pico. <https://www.raspberrypi.com/products/raspberry-pi-pico/>
- [22] ERIC RAYMOND. 2003. The Art of Unix Programming. ADDISON-WESLEY.
- [23] Allison Wang, Rajeev Tomer, and Jack Tench. 2023. InfluxDB: Real-time insights at any scale. <https://www.influxdata.com/>
- [24] Tom White. 2015. Hadoop: The Definitive Guide; storage and analysis at internet scale. O'Reilly et Associates.