EEE3095S Project B - An Environment Logger using a Raspberry Pi and Node Red

Hendri Vermeulen[†] and Kyle du Plessis[‡]
EEE3095S Class of 2019
University of Cape Town
South Africa
[†]VRMHEN004 [‡]DPLKYL002

Abstract

A report detailing the continuation of the environment logger project using a Raspberry Pi and Node Red application to measure and report the sensor data, collected from the environment the Raspberry Pi is placed in, through an online web page for remote access to the information. This implementation allows users to log into a hotspot using a smartphone, view the sensor data and adjust the alarm threshold as well as dismissing the alarm. The stack used for development consists of Python as fixed coding environment, MQTT for data transfer and Node Red to visualise the data.

I. Introduction

The first project, an Environment Logger, was successfully implemented and the research, that the biology student used the Environment Logger for, started gaining ground. More people started to become interested in the research and thus a need for remote access the the systems data aroused.

This report provides an extension to the detailed systems specification of the environment logger project. It essentially serves as a continuation of the initial project and involves the use of two Rapsberry Pi's to present the gathered data to the end users. The first Raspberry Pi works as the environment sensor (as opposed to using Blynk) and transmits the gathered data using a protocol called MQTT (for data transfer) to a node red server which is hosted on the second Raspberry Pi. End users are then able to log-in to a hot spot hosted on the second Raspberry Pi and access the Node Red server to view the data being captured and displayed in a web page. This system implementation enables anyone to access and view the data using a web browser.

The report will cover the design, implementation. testing and results of the system, instructions on how to use the system and the conclusion of the system. The design section will cover the development stack used, design of the Node-Red server, hardware and software interfacing and we'll include a UML Use Case Diagram, UML State Diagram and System Block Diagram to illustrate the design of the system and the expansion of the user requirements. The implementation section will

discuss the methodology we used including steps used to build the system and code snippets to illustrate important code used. The Testing and Results section will cover how we ensured the system works, the functionality we tested and what the results were, screenshots will be included to illustrate our tests and results. The instructions for use section will describe how to use the system including illustrative screenshots with commentary. The last section, conclusion will discuss how well our system did, if we achieve all out objectives for the system and improvements we can make on the system.

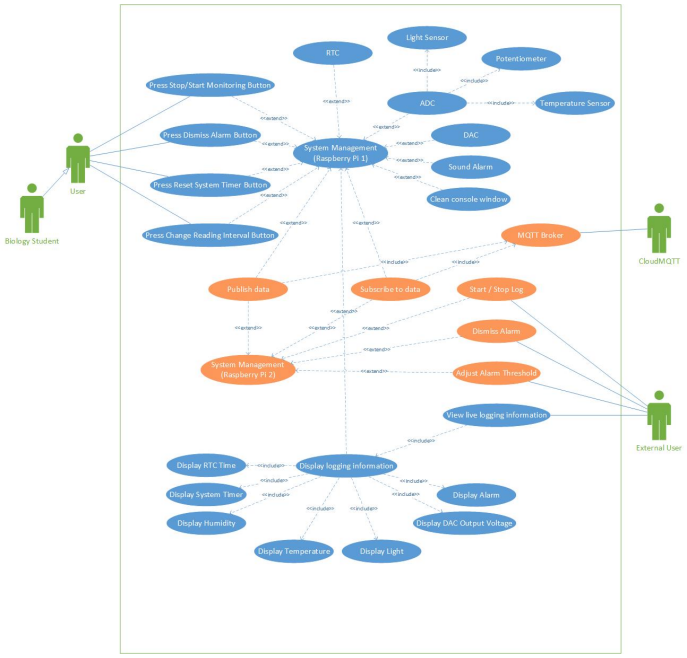
II. Design

This section will first cover a quick discussion on the user requirements expansion and then cover the overall design of the system including design of the server, development stack used, hardware and software interfacing and includes a UML Use Case Diagram, UML State Diagram and System Block diagram to describe the requirements and design of the system.

A. User requirements expansion

The first project, an Environmental logger, led to the biology student's, now on referred to the user, research gaining ground and lead to a need for remote access to the system data so that people interested in the data can access it. Thus, the original use case diagram was expanded to add a second Pi which hosts a server that can be used to access the data, externally. We decided to use Node-Red to host the data server and MQTT Broker for communication between the host server and the environment logger - more discussion on this under System Design section. See use case diagram 1, that depicts this expansion of user requirements, the interactions between the user, Node Red application and the system - raspberry pi and additional hardware.

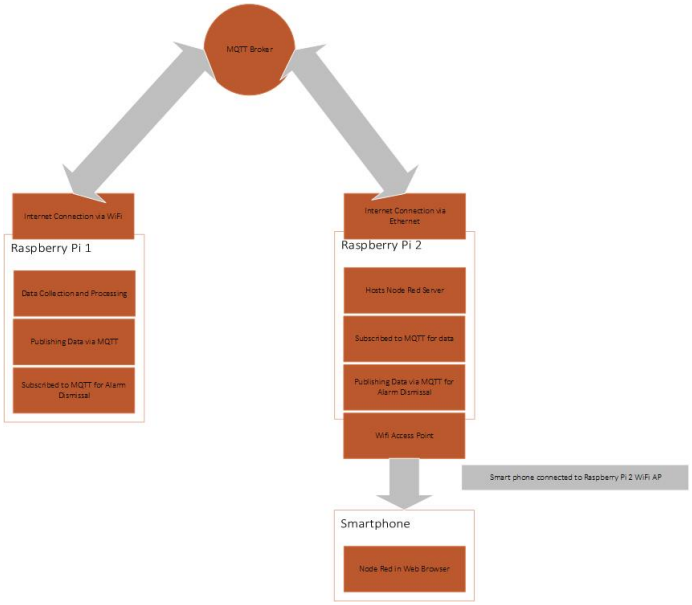
Fig. 1: Use Case Diagram



B. System Design

We require two Raspberry Pi's for this project one to publish, Environment Logger, and another to subscribe to data, remote data access host server for external users to access data. The first Raspberry Pi publishes the gathered data and subscribes to an alarm dismissal signal. The second Pi should publishes an alarm dismissal message and subscribes to the data gathered by the first Raspberry Pi. A web page, Node Red Dashboard, is hosted on the second Raspberry Pi using Node Red and the Raspberry Pi's WiFi interface is turned into an wireless access point. Smart devices, with WiFi capability, are then able to connect to this wireless access point and view the web page reflecting the gathered data. For the MQTT [1] broker, an online publicly available one had been used, namely CloudMQTT [2], to allow the two Raspberry Pi's to communicate over the internet, which allows the two Raspberry Pi's to be in two different locations. A system overview diagram, see figure 2, depicts a high level overview of how the system is set up.

Fig. 2: Block Diagram



C. Development Stack

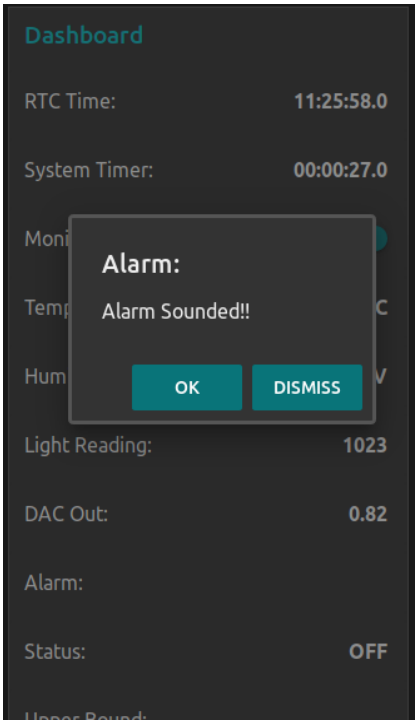
Development stack consists of Python with use of Paho-MQTT library [3], MQTT [1], Node-Red [4] and Node-Red Dashboard [5]. Python had been used as the development language of choice due to its ease-of-programming, extensive libraries, error reduction as well as readability and this makes development much more efficient [6]. Node-RED is an open source visual programming tool used for connecting Internet of Things (IoT) components and allows you to connect nodes together to perform a specific task [4]. It allows for simple and easy MQTT connection establishment between devices, Raspberry Pi GPIO access as well as building responsive graphical user interfaces to view and interact with data. The Node-Red application in this project grabs sensor data gathered by the first Raspberry Pi and visualises the data. MQTT (Message Queuing Telemetry Transport) is a messaging protocol used for Internet of Things applications [1]. MQTT allows for reading and publishing data from sensor nodes which results in easy communication between multiple devices. It is commonly used in publish and subscribe systems where connected devices can publish messages on a specific topic or it can be subscribed to a specific topic to receive messages. The online MQTT broker's function (CloudMQTT [2]) involves receiving and filtering messages then publishing these messages to the respective subscribed clients.

D. Server Design

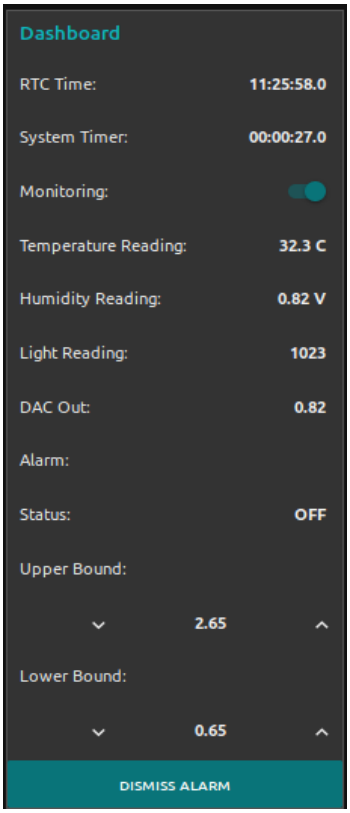
We used Node-Red as the remote data access server and extended it with Node-Red Dashboard for more appealing UI. The server takes in various MQTT subscriptions and displays them in 3 different forms - using 3 different dashboard tabs, in the UI, namely: a standard text form of all the data present, with buttons to dismiss alarm, update thresholds and turn monitoring on and off - see figure 3b for illustration; a gauge form of all the outputs - see figure 3c for illustration; three charts showing the outputs over time - see figure 3d for illustration. These 3 different view of the information present allows the user to pick in which form he would like to see the data. All three of these forms are user-friendly and aesthetically pleasing in their own way. To insure the user always gets notify of the alarm: a notification pop up is always shown when the alarm is triggered no matter which tab the user is in - see figure 3a for illustration. The user can navigate through these tabs using the tab menu on the top left hand corner - see figure 3e for illustration. A server overview diagram, see figure 4, depicts the design of the server. The flow preview shows the configuration of the various MQTT subscribe (input) and publish (output) nodes - available at http://192.168.137.15:1880. The dashboard preview shows the graphical UI design - available at http://192.168.137.15:1880/ui.

Fig. 3: UI Screenshots

(a) UI: Alarm Screenshot

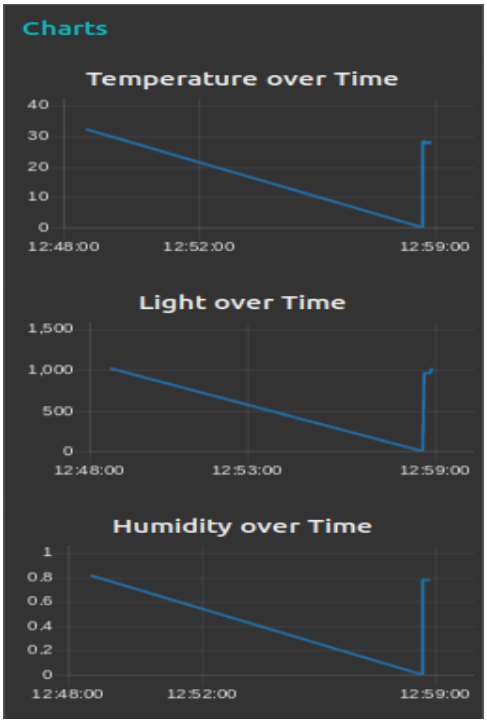


(b) UI: Dashboard (First Tab)



(c) UI: Dashboard Gauges Tab

(d) UI: Dashboard Charts Tab



(e) UI: Dashboard Tab Menu

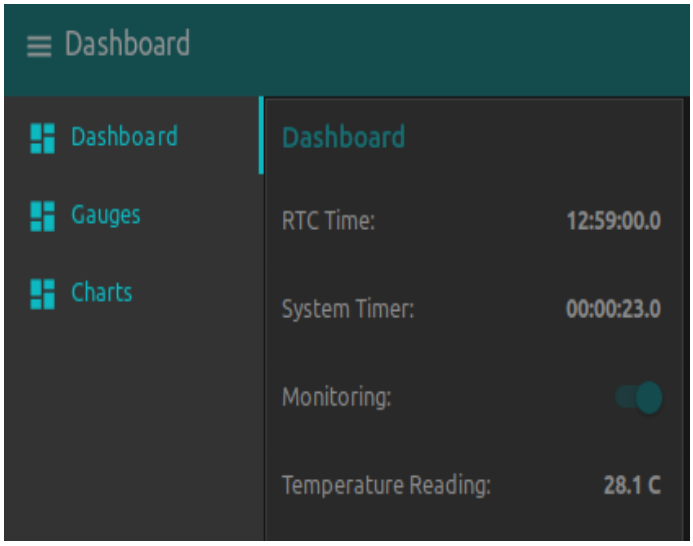


Fig. 4: Node Red Server Flow Design Diagram

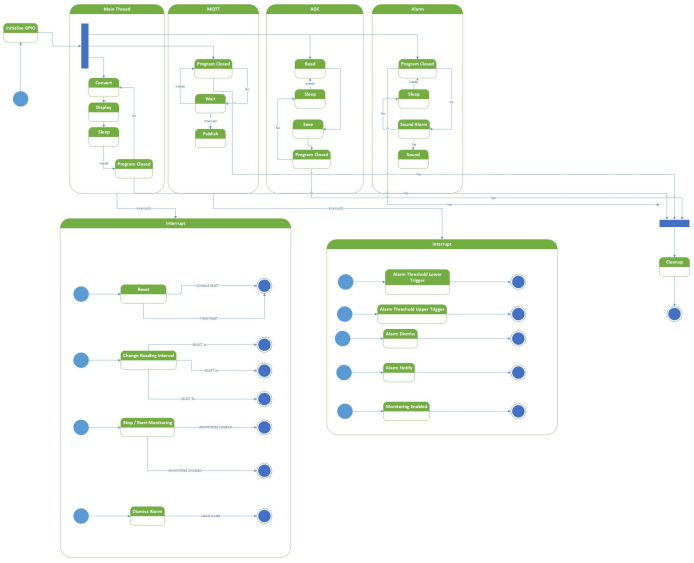
E. Hardware and Software Interfacing

The system need to be remotely accessible and the Node-Red server isn't necessarily going to be in the area then the environmental logger - green house, thus need for networking over the Internet is required to allow the two Pi's to be in separate areas, as long as each area has internet connection. Seeing that the Raspberry Pi's are communicating over the Internet and the system needs to be kept as simple as possible - mitigate the need for use of public ip's and forwarding port in a local area network to insure they can communicate correctly, an online MQTT broker namely: CloudMQTT will be used to handle communication between the two Pi's. This allows simple implementation - Python library Paho MQTT, will handle all networking related communication between the two Pi's, of the communication and no need for setup from the user's side of public ip's - that may continuously change, or port forwarding on their local area network. They just need to insure the two Pi's have internet connection. The external users will be able to connect to the Node-Red hosting Pi via a WiFi hotspot and use their web browser's to see the Node-Red UI, which is automatically handled by Node-Red.

F. System States

The program starts off sequential state, first initializing the program, then it forks into 4 concurrent states: logging the information, updating the ADC sensor values and DAC output voltage, handling MQTT requests and updating the alarm. Each state has composite states where it checks if program has closed and if not, it does its particular work and sleeps or waits for an interrupt - saving CPU time and power. When the program closes all of the states join again into one sequential state where it cleans up and exits. The main thread state, which handles logging of the data, can be interrupted and go into an interrupt state where it does button handling - this state also has composite states where each composite states represents a different button interrupt. The MQTT thread state can also be interrupted into another state - handles its subscription interrupts, also has composite states each depicting a different subscribed topic. All of this is shown in a state diagram, figure 5, depicting the states in which the Python program can be in.

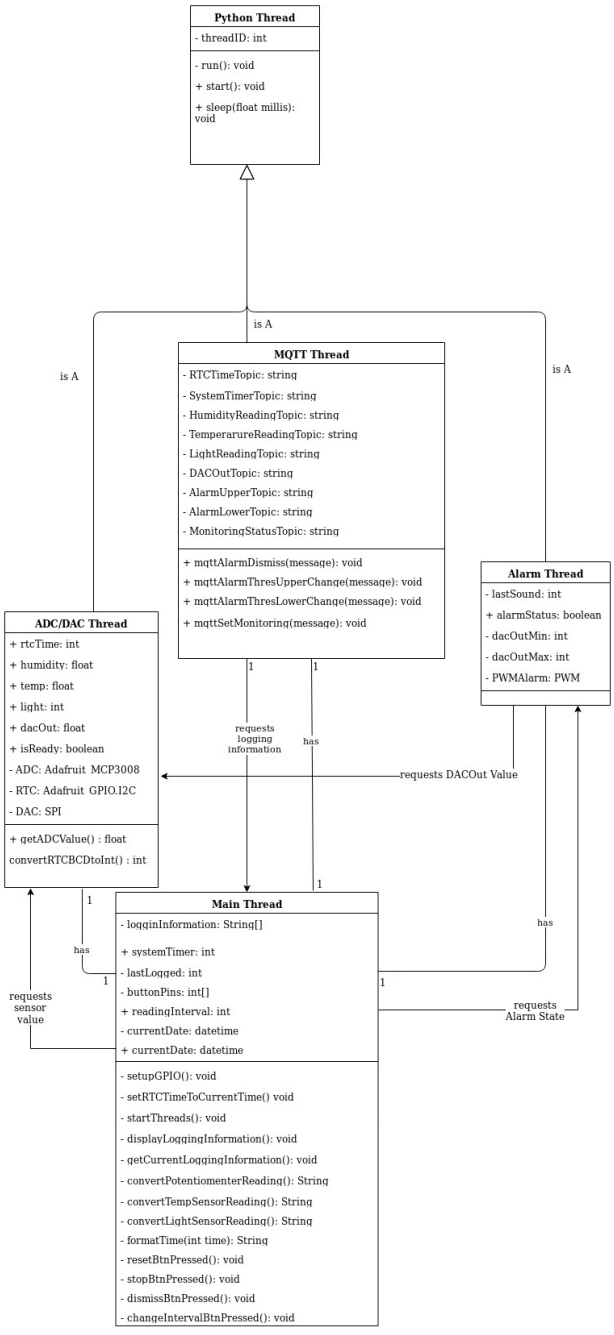
Fig. 5: State Diagram



G. System Structure

The program has four threads: Main Thread, MQTT Thread, ADC/DAC Thread and Alarm Thread. Each thread except the Main Thread is a Python threading thread. They program is mostly structured by threads, each thread has its own function to perform: the main thread handles information logging, converts information from ADC threads saved sensor values to readable values, get alarm status from Alarm Thread, outputs logged information to terminal - each readingInterval second, starts the other threads and waits for the other thread to finish before closing the program with a quick clean up if the program needs to be ended as well as handles button interrupts; MQTT Thread handles communication with the Node Red application and transfers the logged information from main thread to Node Red application; ADC/DAC Thread reads and stores the sensor values as well as updates the DAC output voltage; lastly the Alarm thread check to see when an alarm should be sound, sounds the alarm and also updates the alarm status. See the class diagram, figure 6, depicting the structure of out Python program.

Fig. 6: Class Diagram



III. Implementation

In this section we are going to give a simplified explanation of the implementation as well as explanation and highlights of important code used. We used Python for our implementation due to the extent of libraries available to us on the platform over C.

A. MQTT set up

This code contains the subscribing and publishing implementation using the Paho MQTT Python Client library [3] and the online CloudMQTT broker [2]. The code initiates the MQTT Client, registers Event Handlers and connects to the online CloudMQTT broker. It also initialises various topics and subscribes to particular topics to receive readings that can be displayed for instance on a chart or gauge accessed via a web browser.

```
"""
MQTT
"""

import paho.mqtt.client as mqtt
import time

# initialise variables
MQTT_BROKER = "farmer.cloudmqtt.com" # online broker url
MQTT_PORT = 15215

# onConnect event handler
def onConnect():
    print("Connected to MQTT Broker")

# concurrency lock
lock = threading.RLock()
# used to ensure concurrency between publication and subscription

# initialise MQTT client
mqttc = mqtt.Client()

# register two event handlers
mqttc.on_publish = on_publish
mqttc.on_connect = on_connect

# set cloud MQTT login
mqttc.username_pw_set("lttcflex", "icuxLR9dz3hv")

# connect with online CloudMQTT broker
mqttc.connect(MQTT_BROKER, MQTT_PORT, MQTT_KEEPALIVE_INTERVAL)

# initialise topics
topic_RTCTime = "pi/RTCTime"
topic_SystemTimer = "pi/SystemTimer"
topic_HumidityReading = "pi/HumidityReading"
topic_TemperatureReading = "pi/TemperatureReading"
topic_LightReading = "pi/LightReading"
topic_DACOut = "pi/DACOut"
topic_Alarm = "pi/Alarm"
topic_AlarmThresLowerTrigger = "pi/AlarmThresLowerTrigger"
topic_AlarmThresUpperTrigger = "pi/AlarmThresUpperTrigger"
topic_AlarmNotify = "pi/AlarmNotify"
topic_AlarmDismiss = "pi/AlarmDismiss"
topic_AlarmThresUpperValue = "pi/AlarmThresUpperValue"
topic_AlarmThresLowerValue = "pi/AlarmThresLowerValue"
topic_MonitoringEnabled = "pi/MonitoringEnabled"

# set subscriptions
mqttc.subscribe(topic_AlarmThresLowerTrigger, qos=0)
mqttc.subscribe(topic_AlarmThresUpperTrigger, qos=0)
mqttc.subscribe(topic_AlarmDismiss, qos=0)
mqttc.subscribe(topic_AlarmNotify, qos=0)
mqttc.subscribe(topic_MonitoringEnabled, qos=0)
```

B. MQTT Subscriptions Handling

The program has several subscriptions which, when interrupted, it needs to handle. This is done using the following code:

```
# Dismiss alarm subscription function for button press
def dismissAlarm(arg):
    if (GPIO.input(arg) == GPIO.LOW):
        values["alarm"] = False
        Alarm.ChangeDutyCycle(0)

# Dismiss alarm subscription function for notification button press
def mqttAlarmDismiss(client, userdata, message):
        tmp = str(message.payload.decode("utf-8"))
        if(tmp == "DismissButton" or tmp == "Dismiss"):
                values["alarm"] = False
                Alarm.ChangeDutyCycle(0)

# Alarm threshold change subscription functions
def mqttAlarmThresUpperChange(client, userdata, message):
        global dacVoltMax
        lock.acquire()
        dacVoltMax = float(str(message.payload.decode("utf-8")))
        lock.release()

def mqttAlarmThresLowerChange(client, userdata, message):
        global dacVoltMin
        lock.acquire()
        dacVoltMin = float(str(message.payload.decode("utf-8")))
        lock.release()

# Monitoring change subscription function
mqttLastUpdatedMonitoring = time.time()*1000
def mqttSetMonitoring(client, userdata, message):
        global monitoringEnabled
        global mqttLastUpdatedMonitoring
        currentTime = time.time()*1000
        if(currentTime - mqttLastUpdatedMonitoring > 10): #check each 10ms
                mqttLastUpdatedMonitoring = time.time()*1000
                monitoringSetTo = str(message.payload.decode("utf-8"))
                lock.acquire()
                request = False
                if(monitoringSetTo == "true"):
                        request = True
                monitoringEnabled = request
                lock.release()

#Add MQTT Subscriptions callbacks
mqttc.message_callback_add(topic_AlarmDismiss, mqttAlarmDismiss)
mqttc.message_callback_add(topic_AlarmThresLowerTrigger, mqttAlarmThresLowerChange)
mqttc.message_callback_add(topic_AlarmThresUpperTrigger, mqttAlarmThresUpperChange)
mqttc.message_callback_add(topic_AlarmNotify, mqttAlarmDismiss)
mqttc.message_callback_add(topic_MonitoringEnabled, mqttSetMonitoring)
```

C. MQTT Thread

The MQTT Thread publishes on various topics that have been defined.

```
def publishThread():
    while (not programClosed):  # only continue if parent thread is running
        if (monitoringEnabled):
            publish()
        time.sleep(float(readingInterval))

def publish():
    #Lock to insure not updating subscriptions while publishing
    lock.acquire()

    #Get RTC time and publish
    RTCTime = str(formatTime(values["rtcTime"]))
    mqttc.publish(topic_RTCTime, payload=RTCTime, retain=True)

    #Get System time and publish
    SystemTimer = str(formatTime(systemTimer))
    mqttc.publish(topic_SystemTimer, payload=SystemTimer, retain=True)

    #Get ADC Values and publish
    HumidityReading = str(convertPotentiometer())
    mqttc.publish(topic_HumidityReading, payload=HumidityReading, retain=True)

    TemperatureReading = str(convertTemperatureSensor())
    mqttc.publish(topic_TemperatureReading, payload=TemperatureReading, retain=True)

    LightReading = str(convertLightSensor())
    mqttc.publish(topic_LightReading, payload=LightReading, retain=True)

    #Get DACOut Value and publish
    DACOut = str(getDACOutValue())
    mqttc.publish(topic_DACOut, payload=DACOut, retain=True)

    #Get alarm thesholds and publish
    AlarmUpper = str(dacVoltMax)
    mqttc.publish(topic_AlarmThresUpperValue, payload=AlarmUpper, retain=True)

    AlarmLower = str(dacVoltMin)
    mqttc.publish(topic_AlarmThresLowerValue, payload=AlarmLower, retain=True)

    #Release lock
    lock.release()

    #Check alarm value and publish
    if getAlarmValue() == "*":
        mqttc.publish(topic_Alarm, payload="ON", retain=True)
    else:
        mqttc.publish(topic_Alarm, payload="OFF", retain=True)
```

D. Conclusion of Implementation

Only a high level overview is described of the program implementation and most important code snippets shown to keep report brief. Node-red server implementation was covered in design section, see figure 4. See here for full code: Github Repo Link. See here for demo video: https://www.youtube.com/watch?v=7gVf5m10kag)

IV. Testing and Results

In this section we explain how we ensured our system works, what we did to test the functionality of our system as well as what the results where.

A. Testing

To test the system and its functionality we made use of the following test cases below, see figures 7, 8 and 9. Each test describes the test case number, item testing, description of test, pre-conditions that need to be met before a test can begin, post-conditions that need to be met to pass a test and the inputs and outputs of the test. To ensure the system is working all of these test cases must pass.

Fig. 7: Test Cases, Set 1

Test Case 1
Test Item:
Confirm Node Red is running
Description:
Check that node red server is running on first Raspberry Pi through going to on a laptop connected to it via a web browser and ensure it shows the Node Red Editor as per figure 4
Pre-Conditions:
First Raspberry Pi should have been booted up and has laptop connected to it.
Post-Conditions:
Laptop connected to first Raspberry Pi should have Node Red Dashboard running in it's web browser
Inputs:
URL: , into connected laptop's web browser.
Expected Outputs:
Node Red Dashboard displayed as per figure 4 in connected laptop's web browser.

Test Case 2
Test Item:
Confirm second Pi can connect to server and updates
Description:
Connect the environment logger - second Raspberry Pi, and ensure the node red server gets values from it. This is done by going to on a laptop connected to it via a web browser. This shows the Dashboard UI as per figure 3b. We check that the values are getting updated, by the second Pi, by checking if the timer increases and the readings are as per the terminal output of the second Pi, see figure 10.
Pre-Conditions:
Two Raspberry Pi's should be booted up, have internet connection and are connected to each other via CloudMQTT Broker. Second Raspberry Pi's must be running Project A - Environment Logger, and have associated hardware connected.
Post-Conditions:
The second Raspberry Pi should have terminal output open as per figure 10. The laptop's, that is connected to the first Raspberry Pi, browser should show the Dashboard UI as per figure 3b.
Inputs:
URL: , into laptop's, that is connected to the first Raspberry Pi, web browser.
Expected Outputs:
Node Red Dashboard UI displayed as per figure 3b in laptop's, that is connected to the first Raspberry Pi, web browser. The second Raspberry Pi should display terminal output as per figure 10.

Fig. 8: Test Cases, Set 2

Test Case 3
Test Item:
Confirm get correct values from Environment logger
Description:
To confirm the values are correct as per the environment the logger is in, we change the values: temp, light and humidity; check that terminal updates and check that the node red dashboard (all 3 tabs) updates. 4
Pre-Conditions:
Two Raspberry Pi's should be booted up, have internet connection and are connected to each other via CloudMQTT Broker. Second Raspberry Pi's must be running Project A - Environment Logger, and have associated hardware connected.
Post-Conditions:
The second Raspberry Pi should have terminal output open as per figure 10, and the values should have changed as per beginning of terminal output (changed environment). The laptop's, that is connected to the first Raspberry Pi, browser should show the Dashboard UI as per figure 3b, and should have different values as when it first started.
Inputs:
URL: , into laptop's, that is connected to the first Raspberry Pi, web browser. Change temp by touching temp sensor. Change humidity by changing potentiometer to all the way up and all the way down. Change light reading by covering light sensor with thumb and shining a phone torch on it.
Expected Outputs:
Node Red Dashboard UI displayed as per figure 3b in laptop's, that is connected to the first Raspberry Pi, web browser. The second Raspberry Pi should display terminal output as per figure 10. The values displayed in the terminal and dashboard should change as we give different input as per input section of this use case.

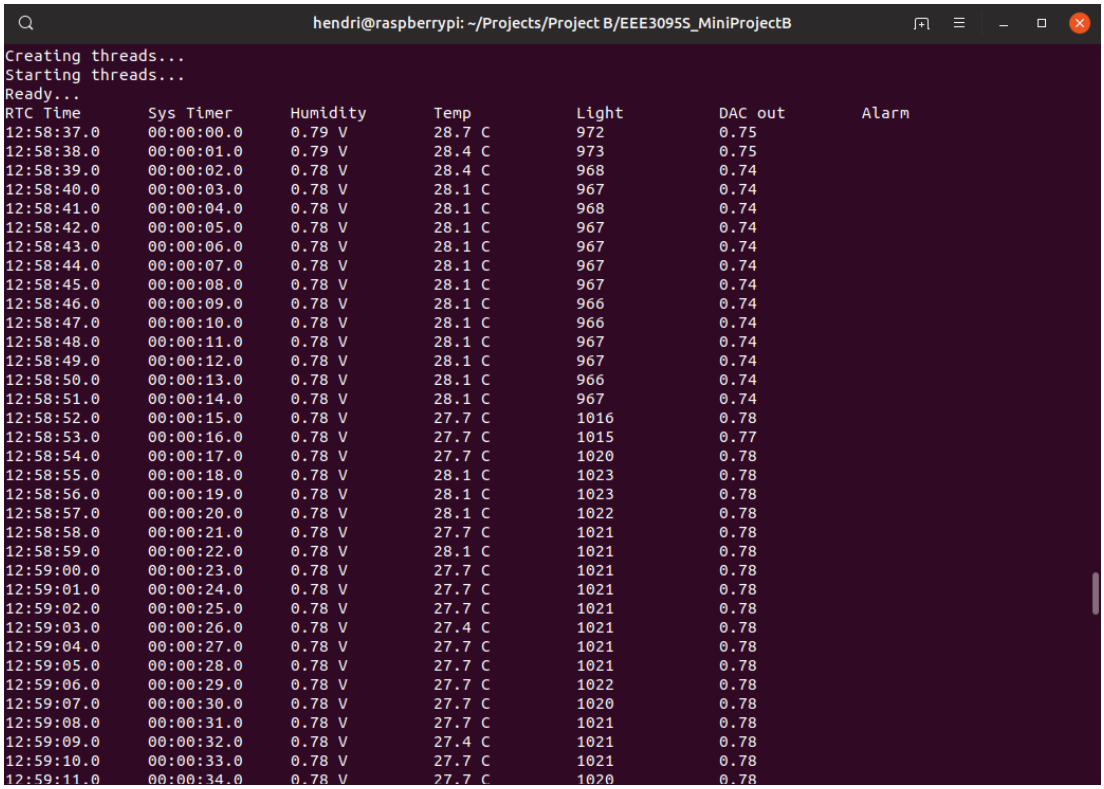
Test Case 4
Test Item:
Confirm External User can connect
Description:
Lastly we need to check that an external user can connect to the server and see logs. This is done by connecting a mobile device via WiFi to the first Pi's WiFi hotspot, SSID: VRMHEN004Pi Password: fruitloops, and it can go via it web browser to and see the same values as seen in step 2, step 3 is also repeated to ensure external user gets latest and correct values.
Pre-Conditions:
Two Raspberry Pi's should be booted up, have internet connection and are connected to each other via CloudMQTT Broker. Second Raspberry Pi's must be running Project A - Environment Logger, and have associated hardware connected. First Raspberry Pi should have WiFi Hotspot on and a phone should be connected to that hotspot.
Post-Conditions:
The second Raspberry Pi should have terminal output open as per figure 10. The phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, browser should show the Dashboard UI as per figure 3b.
Inputs:
URL: , into phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, web browser.
Expected Outputs:
Node Red Dashboard UI displayed as per figure 3b in phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, web browser. The second Raspberry Pi should display terminal output as per figure 10.

Fig. 9: Test Cases, Set 3

Test Case 5
Test Item:
Dismissing alarm and updating thresholds
Description:
After confirming that the Pi's can connect to each other and communicate accurately and that an external user can observe these values, we need to check if an external user can also dismiss the alarm. This is done by triggering the alarm - increase or decrease humidity via Potentiometer or increase light at the second Pi, then seeing that the status of the alarm, next to status label in Dashboard Tab - see figure 3b, is updated to "ON" and the notification - see figure 3a, is displayed. Then we did two tests: one: ignore the alarm notification by pressing "OK" and using the dismiss button - located at the bottom of Dashboard Tab, then checking if alarm status is updated to OFF; two: pressing "Dismiss" on the alarm notification (in all 3 tabs) and checking if the alarm status is updated to "OFF".
Pre-Conditions:
Two Raspberry Pi's should be booted up, have internet connection and are connected to each other via CloudMQTT Broker. Second Raspberry Pi's must be running Project A - Environment Logger, and have associated hardware connected. First Raspberry Pi should have WiFi Hotspot on and a phone should be connected to that hotspot.
Post-Conditions:
The second Raspberry Pi should have terminal output open as per figure 10. The phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, browser should show the Dashboard UI as per figure 3b. The alarm should be off after test.
Inputs:
URL: , into phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, web browser. Change the potentiometer reading up or down to activate alarm. Change thresholds up or down in Dashboard, on phone, to check alarm only triggers when inside the thresholds. Press dismiss button in alarm notification in all 3 tabs. Press dismiss button in first tab.
Expected Outputs:
Node Red Dashboard UI displayed as per figure 3b in phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, web browser. The second Raspberry Pi should display terminal output as per figure 10. The alarm status should change as alarm is triggered and turned off by notification dismiss button or dismiss button under first tab. Thresholds should change value as they are being updated and alarm notification should appear when DACOut goes outside the thresholds.

Test Case 6
Test Item:
Enabling and Disabling Monitoring
Description:
Lastly we checked if an external user can enable or disable the monitoring off the Environment logger, this was done by switching the switch next to the "Monitoring" label under Dashboard Tab, see figure 3b, and ensuring the output in the terminal of the second Pi stops, see figure 10, and that the timers in the Dashboard Tab stops incrementing. Then we enable again and check if the terminal output on the second Pi and the timers on the Dashboard Tab starts again.
Pre-Conditions:
Two Raspberry Pi's should be booted up, have internet connection and are connected to each other via CloudMQTT Broker. Second Raspberry Pi's must be running Project A - Environment Logger, and have associated hardware connected. First Raspberry Pi should have WiFi Hotspot on and a phone should be connected to that hotspot.
Post-Conditions:
The second Raspberry Pi should have terminal output open as per figure 10. The phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, browser should show the Dashboard UI as per figure 3b. Monitoring should be off on the Dashboard - phone's browser, and in terminal - second Raspberry Pi.
Inputs:
URL: , into phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, web browser. Press the switch in the Dashboard, on the phone, as per figure 3b.
Expected Outputs:
Node Red Dashboard UI displayed as per figure 3b in phone's, that is connected to the first Raspberry Pi's WiFi Hotspot, web browser. The second Raspberry Pi should display terminal output as per figure 10. These outputs should stop updating when monitoring is switch off and start again when monitoring is switch on again.

Fig. 10: Environment Logger Terminal Output



B. Results

The server and WiFi hotspot are always running, starts on boot up, as long as first Pi is power on and booted up - thus Test Case 1 and 2 are always successful. Test Case 3 and 4 resulted in fast, responsive and correct updating of values and always has values that relate to the actual environment. The last two Test Cases always resulted in fast, responsive and concurrent execution. The two Raspberry Pi's where always synchronised and everything happened in real-time. No test cases failed, thus all system and functionality tested was successful.

V. Introduction on how to use

Here are the instructions on how to use this system. There are various sub-sections each describing how to use a certain function of the system. The first few sections are on how to start and connect to the server.

A. Start the Node Red Server

The node red server starts up when the Raspberry Pi boots - thus, there is no need to start the server manually.

B. Connect to and Access the server via WiFi

To connect the server using WiFi, search for VRMHEN004Pi SSID on your smart device. Once found connect with the password: fruitloops. Once the WiFi is connected you can access the server's UI via your web browser and the following URL: . Once you have entered this URL on your web browser you should see figure 3b displayed in your web browser.

C. Navigation through the various tabs

Once you have connected to the server and accessed the server through WiFi and your web browser as per the previous section: you should see a menu option on the top left hand corner of your screen, press it to display the navigation menu, see figure 3e for illustration. This will show you 3 options: Dashboard, Gauges and Charts - these are the three tabs and three ways of how the information of the Environment Logger is displayed.

D. Dismissing the Alarm

There are two options to dismiss the Alarm, one: via the Dashboard Tab, see figure 3b, there is a button at the bottom of the page with the label: "Dismiss Alarm" - press it to dismiss the alarm; two: when the alarm is sound you should see a notification as per figure 3a, press the "Dismiss" button to dismiss alarm or OK button to ignore it - can dismiss alarm still using option one.

E. Updating the alarm thresholds

The thresholds of the alarm can be updated using increment and decrements buttons under the label Alarm in the Dashboard Tab, see figure 3b. When you increment or decrements these values they are automatically updated.

F. Enable/Disable Monitoring

You can enable or disable monitoring using the switch button next to the label: "Monitoring:" under the Dashboard Tab, see figure 3b, press it to switch between enabled (blue - circle to the right) and disabled (gray - circle to the left).

VI. Conclusion

In conclusion, all the user requirements were met and all objectives were achieved, whilst keeping the system within real-time performance using multi-threading without any concurrency issues: The Node-Red Server and Wifi Hotspot always runs on boot to allow external users anytime access to the live data and allow the two Raspberry Pi's to always be able to communicate as long as there is Internet connection - thanks to the CloudMQTT MQTT Online Broker. The system provides all the required functionality at high performance and passed all tests during validation process: The system was always fast, responsive and had correct environmental logging information on both Raspberry Pi's in real-time.

Although the system is already fully-functional and performing in accordance with user requirements, the system can still be improved by using C++ as a development language to further increase system run-time performance and thus ensuring even more real-time performance [7]. Furthermore the system can be improved with user authentication to allow only system administrators or authorised external users to dismiss alarm, enable/disable monitoring or change alarm threshold as at this time any user has access to the full system. Currently the system is only accessible to external users through the Node-Red hosting Pi's WiFi hotspot thus, the external user must be in WiFi range of the Node-Red hosting Pi. This isn't particularly convenient and can be improved by allowing users to connect to the Node-Red Dashboard via the Internet instead of a local network. However, then the previous mention of user authentication is needed to ensure only authorised users have access to critical-use parts of the system - disable/enable monitoring or alarm. Furthermore, a mobile application could be created, using appropriate UI/UX design principles, for both Android and iPhone smartphones as an alternative way of presenting the data to end users.

REFERENCES

[1] Ciro, R. Santos, John, E. Gongora, O. Scholten, Huong, Andre, S. Santos, B. Calder, Martin, and et al., "What is mqtt and how it works," Apr 2019. [Online]. Available: https://randomnerdtutorials.com/what-is-mqtt-and-how-it-works/

[2] "A globally distributed mqtt broker." [Online]. Available: https://www.cloudmqtt.com/

[3] "paho-mqtt." [Online]. Available: https://pypi.org/project/paho-mqtt/

[4] "Red." [Online]. Available: https://nodered.org/

[5] "red-dashboard." [Online]. Available: https://flows.nodered.org/node/node-red-dashboard

[6] "The rise of python for embedded systems continues," Dec 2018. [Online]. Available: https://www.zerynth.com/blog/the-rise-of-python-for-embedded-systems-continues/

[7] W. G. Wong, "Is c the best embedded programming language?" Feb 2018. [Online]. Available: https://www.electronicdesign.com/embedded-revolution/c-best-embedded-programming-language