UNIVERSITY OF SCIENCE

ADVANCED PROGRAM IN COMPUTER SCIENCE

NGUYEN THANH TUNG

A SECURE, SCALABLE, FOTA-CAPABLE, HOME AUTOMATION IOT SYSTEM TO CONTROL FAN AND AIR-CONDITIONER ELECTRONICS

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

HO CHI MINH CITY, 2021

UNIVERSITY OF SCIENCE

ADVANCED PROGRAM IN COMPUTER SCIENCE

NGUYEN THANH TUNG – 1751119

A SECURE, SCALABLE, FOTA-CAPABLE, HOME AUTOMATION IOT SYSTEM TO CONTROL FAN AND AIR-CONDITIONER ELECTRONICS

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

THESIS ADVISORS

Dr. TRAN THANH PHUOC

HO CHI MINH CITY, 2021

Acknowledgement

First and foremost, I would like to express my deepest gratitude towards my two mentors and supervisors, Dr. Tran Thanh Phuoc and M.S. Huynh Thanh Tu. You have given me all the guidance and motivation I needed to complete this work. I am truly blessed to be under your tutelage for my bachelor thesis.

Second, I would like to express my sincerest gratitude towards my parents. You have always showered me with overwhelming love and support throughout my study career.

Finally, I would like to thank my friends. You were always there for me. Your tremendous emotional support gave me just enough to push through all the hardships.

UNIVERSITY OF SCIENCE
ADVANCED PROGRAM IN COMPUTER SCIENCE

Thesis Proposal

Thesis title:

A secure, scalable, FOTA-capable, home automation IoT system to control fan and air-conditioner electronics

Thesis advisor: Dr. Tran Thanh Phuoc (Ton Duc Thang University)

Students: Nguyen Thanh Tung (1751119)

Type of thesis: (*Technology with demo application*)

Duration: (From *Jan, 2021 to July, 2021*)

Contents of thesis:

To propose an IoT home automation application to adjust the indoor atmosphere. The application would do so by controlling fan and AC electronics. There are two things we hope to achieve. First, we want our device control method to be non-intrusive. We will use infrared communication to control the devices. Furthermore, because infrared signals may be lost in transmission, we will also implement a control feedback mechanism. Second, we aim to address as much of the popular IoT challenges as possible. Those are the challenges of security, scalability, updatability, networking and automation. More specifically, we seek to make our application secure at every network level, dynamically scalable and capable of performing firmware-over-the-air (FOTA) updates. There are two types of nodes in our IoT system, a controller node on the Raspberry Pi hardware and a sensor/actuator node on the ESP32 hardware. The nodes will communicate using Wi-Fi technology. Regarding networking, we will carefully set up the Raspberry Pi as a wireless access point for ESP32 that can provide internet capability when the home Wi-Fi router is up. The Raspberry Pi also has to host a DNS service for TLS security to be possible. The whole system would be hidden in a private network, but remote access must still be available. At the application level, the nodes will communicate using the MQTT publish-subscribe

protocol. Regarding automation, the controller node will do all the processing, while the sensor/actuator node will only publish sensor data and receive fan and AC commands. We will have to research methodologies to convert temperature and humidity values into a comfortability heuristic for use within our application. The application will further combine that heuristic with lighting condition, human present condition, etc., to automatically derive control commands for the fan and AC in order to improve the room atmosphere. The sensor/actuator node will be programmed in MicroPython, while the controller node will be built using the Node-red flow-based visual programming tool. A web UI will also be provided to the end user. Through the UI, the user should be able to improve the room condition and directly control the fan and AC in manual mode, or let the system take control in automatic mode.

Research timelines:

- Researching related work
- Researching the necessary tools and technical knowledge
- Setting up network functions on Raspberry Pi
- Testing with the Node-red tool
- Testing with the MQTT protocol
- Securing Node-red, HTTP and MQTT
- Testing with infrared communication
- Designing the core application model to be implemented
- Implementing the sensor/actuator service on ESP32
- Implementing the controller service in Node-red on Raspberry Pi
- Implementing the web UI
- Testing and debugging the system
- Enabling remote access
- Making the system scalable
- Writing the thesis

Approved by the advisor	**Ho Chi Minh city, Mar/19/2021**
Signature of advisor	***Signature(s) of student(s)***

A secure, scalable, FOTA-capable, home automation IoT system to control fan and air-conditioner electronics

Featuring MQTT protocol, MicroPython and Node-red

Contents

1. Problem statement 1
2. Related work 3
3. Overview 4
4. Acronyms and abbreviations 6
5. IoT Design 8
   - 5.1. IoT design essentials 8
   - 5.2. MQTT essentials 9
   - 5.3. Top module 10
   - 5.4. Temperature submodule 12
   - 5.5. Lighting submodule 12
   - 5.6. Motion submodule 13
   - 5.7. Wind submodule 14
   - 5.8. Fan submodule 15
   - 5.9. AC submodule 16
   - 5.10. MQTT submodule 18
   - 5.11. Infrared design 19

5.12. Routing 21
6. Automation 22
6.1. Node-red design overview 22
6.2. Heuristic selection 24
6.3. Automatic control 25
7. Scalability 26
8. Networking 27
8.1. LAN configuration 27
8.2. Edge case handling 29
9. Security 30
9.1. Network Level 30
9.2. Transport Level 30
9.3. Application Level 31
10. Updatability (FOTA) 32
10.1. The FOTA flow 32
10.2. Security 35
10.3. Safety 36
11. Web UI 38
11.1. Tab menu 39
11.2. Connection 40
11.3. Version 40
11.4. Control 41
11.5. Level 41
11.6. Temperature 42
11.7. Humidity 43
11.8. Comfortability 44
11.9. Lighting 45

11.10. Motion 45
11.11. Human 46
11.12. Fan 47
11.13. Aircon 48
11.14. Preference 49
11.15. Reset 49
11.16. FOTA Update 50
1. Result 50
2. Limitations 51
13.1. IoT limitations 51
13.2. Automation rules limitations 52
13.3. Scalability limitations 52
13.4. Remote access limitations 52
13.5. Updatability (FOTA) limitations 52
13.6. Usability limitations 53
13.7. Other limitations 53
3. Conclusion 54
4. Future work 54

List of tables

Table 4.1. Table of acronyms and abbreviations ……..... .....……..6

List of figures

Figure 3.1. The IoT system topology …….....……..4
Figure 5.1. The esp32 IoT service module hierarchy …….. .....……10
Figure 5.2. DHT22 sensor ……... .....……12
Figure 5.3. CDS - NVZ1 sensor …….. .....……12
Figure 5.4. PIR HC-SR501 sensor …….....……13
Figure 5.5. YJ-FS RS232/RS285/TTL sensor ……. .....……14
Figure 5.6. Magnetic door switch sensor …….....……17
Figure 5.7. TSOP1838 IR receiver …….. .....……19
Figure 5.8. Generic IR transmitter LED …….. .....……19
Figure 5.9. Routing schematic …….....……21
Figure 6.1. The RPi automation controller in a Node-red flow …..…….....……22
Figure 6.2. Dew point and comfortability chart …….. .....……34
Figure 8.1. Summary of the LAN setup process ……. .....……34
Figure 11.1. The Web UI when most components are turned on ….. .....……38
Figure 11.2. UI Tab menu …….. .....……39
Figure 11.3. UI Connection panel …….....……40
Figure 11.4. UI Version panel …….....……40
Figure 11.5. UI Control panel …….....……41
Figure 11.6. UI Level panel ……... .....……41

Figure 11.7. UI Temperature panel ……42
Figure 11.8. UI Humidity panel ……43
Figure 11.9. UI Comfortability panel ……44
Figure 11.10. UI Lighting panel ……55
Figure 11.11. UI Motion panel ……55
Figure 11.12. UI Human panel ……56
Figure 11.13. UI Fan panel ……57
Figure 11.14. UI Aircon panel ……58
Figure 11.15. Preference panel ……59
Figure 11.16. Reset panel ……59
Figure 11.17. Update UI Tab and FOTA menu ……50

List of algorithms

Algorithm 5.1. Pseudocode for the pin callback function of motion submodule 14
Algorithm 5.2. Pseudocode for the routine function of wind submodule ……...15
Algorithm 5.3. Pseudocode for routine function of fan submodule ……... ...16
Algorithm 5.4. Pseudocode for routine function of Ac submodule ……...18
Algorithm 5.5. Pseudocode to capture infrared signal …….....20
Algorithm 10.1. Pseudocode for the FOTA update process …….. ...34
Algorithm 10.2. Pseudocode for secure checking of the latest firmware version …. ...36
Algorithm 10.3. Pseudocode for secure downloading of a firmware folder …….....36
Algorithm 10.4. Pseudocode for secure downloading of a firmware file ……...37

Abstract

The thesis demonstrates an all-in-one internet-of-things (IoT) model to control fan and air-conditioner (AC). Our ultimate aim is to propose a well-incorporated IoT solution that tackles as many common IoT challenges as possible. Those are the challenges of security, scalability, updatability, networking and automation.

To begin with, first, regarding security, we put our system under three layers of protection. Those are protection at the network level, transport level and application level.

Second, regarding scalability, we can scale our unit of control, which is one fan with one AC, to any size automatically without any reconfiguration.

Third, regarding updatability, we examined an open-source implementation of over-the-air firmware update (FOTA) and modified it extensively to suit our functional and safety requirements.

Fourth, regarding networking, we use the MQTT protocol for communication between the nodes in our carefully configured local area network (LAN). Not only that, we had to ensure safe operation of the nodes in every connection edge-case, such as when LAN is not available, or when it is but without internet capability.

Fifth, regarding automation, we will explain how we chose a comfort level heuristic and derive automatic control rules. Node-red, the popular flow-based visual programming tool, is highly suitable for such a programming task.

Finally, we choose consumer infrared (CIR) as our target of application. From the start, we were looking for a non-intrusive method to introduce automation to the current household appliances, and infrared remote control was such a method. Among typical CIR home electronics, we choose to work the fan and AC, as they conveniently share the same purpose of refining the surrounding atmosphere.

1. Problem statement

The IoT community is in need of a lot of things. They need a proposal for a full-flow IoT system. They need a proposal that is high in topic coverage. They need a proposal that is rich in implementation details. They need a proposal that tells how to turn houses into smart homes without all that excessive instrumentation. Ultimately, they need a proposal that effectively addresses one of the most prominent IoT goals, human's comfortability. The reason they are in need of all of these is because of the myriad of merits such solutions bring about. First, we identify the benefits of a full-flow IoT solution. A full-flow model would give us an all-inclusive view of the design. Since no development stage is omitted from the design, the system is highly guaranteed to be a practical, functioning one. A full-flow model also provides valuable insights into how to deal with some integration problems, or rarely seen problems resulting from interactions of different components. You will also develop a more coherent view of the system, since every part of the system has to serve a common design goal or purpose. Second, we explore the benefits of an all-in-one solution that seeks to deal with as many IoT challenges as possible. Such a solution would inherit all the individual reasons that make each challenge worth tackling. For example, tight security is needed because it is of utmost importance to human safety, scalability for the ease of mass installation and maintenance, updatability for the system's ability to cope with bugs and evolve. That is not to mention the effect of a flexible, loosely-coupled data network, or automation being the primary development challenge of modern society. A comprehensive IoT solution would also prove to be a very competitive one, as it tries to meet standards in each and every aspect possible. Third, we look into the benefits of an IoT solution that is rich in implementation details. A sufficiently specific model would be easier to grasp and more practical. If it is specific

enough, it could also serve as learning materials for IoT beginners to take on the field. Furthermore, it would also help to increase the success rate of re-implementation of the system under the hands of others. Fourth, we discuss the benefits of a non-intrusive smart home renovating solution, which is rarely seen in other works. Compared to an intrusive solution that attempts to alter the natural circuits of household electronic devices, a non-intrusive one is often fail-safe. You can always revert back to the original device control scheme lest something undesirable happens. It would also be beneficial for the IoT developers to be able to perform any number of trial-and-errors on the controlled devices without the fear of permanently damaging or corrupting them. Infrared communication is a non-invasive technique to control electronics. Not only that, household infrared electronics are very common. Hence, we choose infrared as our target IoT field of application. Fifth and finally, we concern ourselves with delivering human comfortability. Such is among the most central IoT smart home goals, the others being energy conservation and human safety. One of the most direct and impactful ways to elevate indoor comfortability is to automatically control the indoor atmosphere. Fortunately for us, there is the fan and the air-conditioner. They are ubiquitous infrared devices. They also conveniently share the same purpose of improving the indoor atmosphere. Putting the fan and air-conditioner under automated control would truly help solving the comfortability problem. In essence, we realise the substantial benefits and propose an IoT solution that addresses all five of the points mentioned above. That is a full-flow, topic-comprehensive, implementation-detailed, non-intrusive and impactful IoT solution. The proposed model would automatically control the fan and air-conditioner infrared devices to make indoor air comfortable. But since infrared signals may sometimes fail to reach the target, we also have our infrared control feedback mechanism.

2. Related work

First of all, various smart home technologies, including GSM, Bluetooth, IoT, PIC microcontroller with ZigBee modulation are discussed at [1]. Regarding IoT smart home, prominent IoT technologies and applications in the field of home automation, which are on energy savings, human safety and comfortability, are discussed at [2]. We can find really interesting work involving machine learning in IoT home safety at [3]. Next, we have to consider the connectivity standards and challenges. Many, among WiFi, ZigBee, Z-Wave, Bluetooth LE and Thread technologies are discussed at [4]. Also, a really flexible and novel connection framework is discussed at [5]. Eventually, we would settle on the MQTT communication protocol for its loose-coupling, plug-and-play traits. Besides, the publish-subscribe model of MQTT has some really attractive features [6]. There are some typical and effective IoT approaches. One such implementation is on the ability to remotely control all sorts of household devices [7]. Another is on basic ON/OFF controls of multiple appliances in a circuit intrusive fashion, also with provided circuit schematics [8]. We personally prefer a non-intrusive method. There is also work that is similar to ours that uses infrared application but without concerns about control feedback [9]. Next, we have to look into some methodologies that process temperature and humidity values to manipulate the room atmosphere. A fine method using fuzzy logic, although applied to the computer server room, is discussed at [10]. We go on to examine typical security risks and vulnerabilities. A thesis that gives inclusive details about numerous IoT challenges and measures can be found at [11]. Following are some related work about the firmware-over-the-air (FOTA) update feature. Demonstration of a certain FOTA FOTA procedure with some algorithmic details is available at [12]. Various FOTA attacks are discussed at [13], and a wide range of FOTA constraints and techniques are discussed at [14]. Next, we focus on the scalability aspect of an IoT system. A

sufficient amount of scalability requirements can be consulted at [15]. After that, we also look at the usability aspect of the system. An IoT UI design detailed in elements and aspects is present at [16]. Finally, some machine learning ideas worth implementing for future work are also explored. Two of them are machine learning work on detecting malfunctioning IoT nodes for operation safety [17] and work on integrating human activity recognition into IoT for smarter automation control [18].

3. Overview

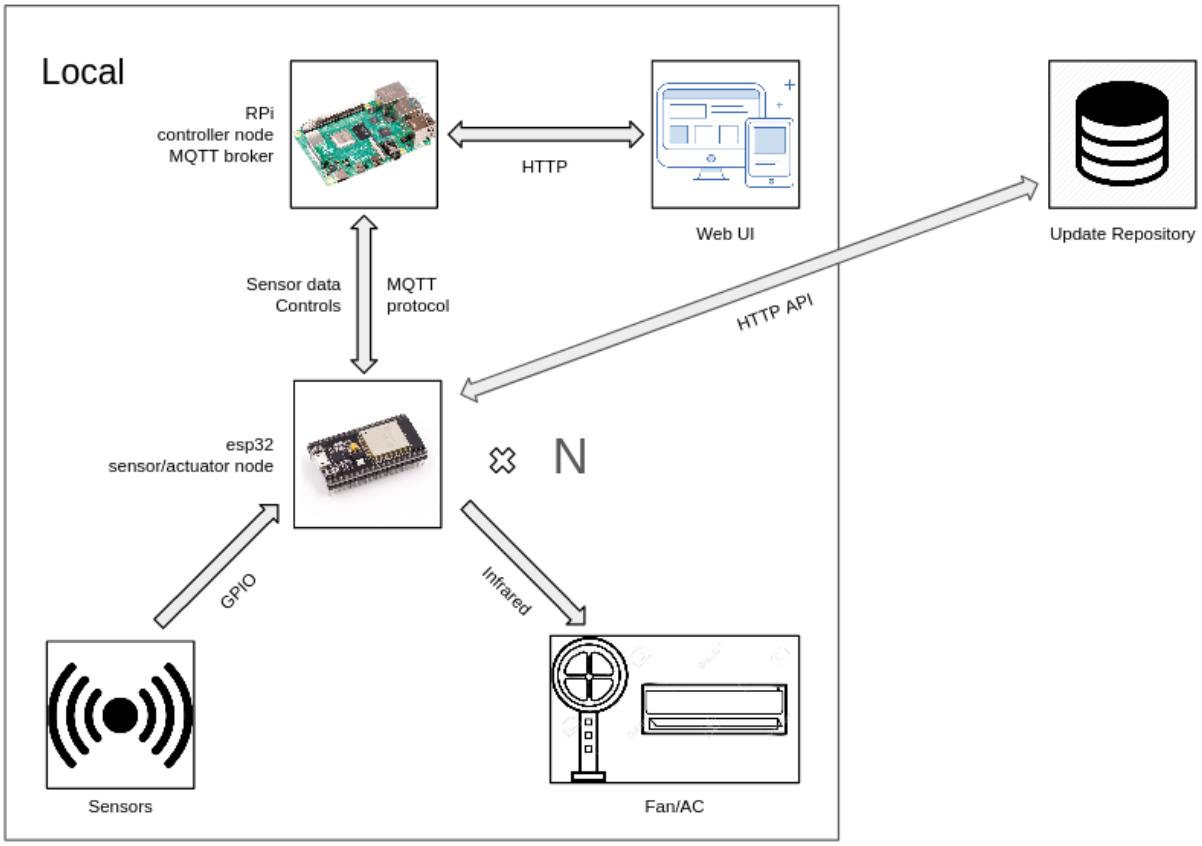

Figure 3.1. The IoT system topology

Just as an IoT system is often composed of various types of nodes, our model consists of two types of nodes, which are the controller node and the sensor/actuator node. A controller node's service is hosted on a Raspberry Pi (RPi) 4 Model B hardware. On the other hand, the sensor/actuator node offers its service on a ESP32-WROOM-32. For simplicity purposes, we will refer to each of these nodes by the board on which they are situated. In this case, the controller node will simply be referred to as RPi, while the sensor/actuator node is referred to as esp32. The esp32 is physically connected with external hardware sensors and IR transmitters. Its sole purpose is to provide sensor data, along with a device control interface to the RPi. Alternatively, RPi plays the role of a main controller unit, and is often coupled with multiple esp32. It processes the received sensor data, then determines the right policy to follow from the automatic control rules, eventually sending instructions back to esp32 to be carried out. When the esp32 receives commands that set the desired fan/AC state, it checks if the current fan or AC power state is the desired power state. If it is, the esp32 simply does nothing. If it is not, esp32 will immediately emit the ON/OFF toggling signal. It then waits for the sensor feedback to determine if the operation is a success. In case the IR signal fails to reach the target device, the operation will be repeated until the desired fan or AC state is finally attained. That is our infrared feedback control mechanism. Users can interact with the system through a web user interface (UI). The interface provides the user with the ability to track temperature and humidity value, check for human presence and lighting condition, monitor power states of the fan and AC. On top of that, administration of the system can easily toggle between automatic and manual control. In automatic mode, only the reset and firmware update options are available, while in manual mode, the user can additionally set the desired state of the fan and AC without intervention from the system. All programming on the Raspberry Pi, including the web UI, is performed using the Node-red flow-based visual programming tool. Meanwhile, the complete IoT service on ESP32 is coded in

MicroPython, a port of the popular interpreted programming language Python. Data exchange between esp32 and RPi is done using MQTT, which is a publish-subscribe network protocol. According to Wikipedia [19], “The MQTT protocol defines two types of network entities: a message broker and a number of clients. An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients”. In this case, the MQTT clients are the esp32 and the Node-red portion of RPi, while the MQTT broker is hosted as a Linux service on RPi, alongside the Node-red daemon. The whole system network is hidden in a private local network provided by the RPi access point, behind the home router NAT. However, remote access is still possible through a tunneling technique. Furthermore, inside the private network, HTTP and MQTT exchanges are all secured under Transport Layer Security (TLS). Lastly, both the Node-red and the MQTT broker enforce user authentication and authorization at the application level.

4. Acronyms and abbreviations

Abbreviation	Definition
AC	air-conditioner
AP	access point
API	application programming interface
esp32	ESP32-WROOM-32
CIR	consumer infrared
DNS	domain name system

IoT	internet-of-things
GPIO	general-purpose input/output
IR	(consumer) infrared
LAN	local area network
NAT	network address translation
FOTA	firmware-over-the-air
P2P	peer-to-peer
PC	personal computer
RAM	random access memory
REPO	repository
REST	representational state transfer
RPi	Raspberry Pi
TLS	transport layer security
UI	user interface
URL	uniform resource locator
WNIC	wireless network interface controller
VPN	virtual private network

Table 4.1. Table of acronyms and abbreviations

5. IoT Design

5.1. IoT design essentials

The IoT service on esp32 is programmed in MicroPython, a port of the interpreted language Python that is optimized to run on a microcontroller. Its standard syntax is mostly similar to Python as well. The standard library is also reimplemented, although with a few subtle differences. Furthermore, there are many MicroPython-specific library modules, even portions that are specific to different microcontroller boards. You can refer to [20] for the full MicroPython API documentation. The esp32 has general-purpose input/output (GPIO) pins. These pins can be read and driven using the language application programming interface (API). GPIO pins are used to connect esp32 to external hardware sensor and actuator modules. Most sensor and actuator modules have a simple data interface. They often output logic 1 to represent a semantically assigned ON state, logic 0 to represent the OFF state, or the opposite if the data pin is active low. Each sensor/actuator typically requires a pair of power and ground connections. The power pin is used to drive the logic 1 (pull-up) network, while the ground pin is used to drive the logic 0 (pull-down) network in a hardware module. However, if a module lacks one out of the two connections mentioned above, take for example, it lacks a power connection, then the module can only drive logic 0, and vice versa. In another word, we say that the module lacks the pull-up, or pull-down capability, respectively.

5.2. MQTT essentials

For the sake of completeness, we will have a brief description about the MQTT protocol. Since the Wikipedia page of MQTT has put it really well, we might as well cite them here. According to Wikipedia [19],

"The MQTT protocol defines two types of network entities: a message broker and a number of clients. An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients. An MQTT client is any device (from a micro controller up to a fully-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.

Information is organized in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The publisher does not need to have any data on the number or locations of subscribers, and subscribers, in turn, do not have to be configured with any data about the publishers."

At the core, there are four types of MQTT messages, which are the *connect*, the *subscribe*, the *publish* and the *disconnect* message. The *connect* message requests a connection to be established between the client and the broker. The *subscribe* message registers a topic subscription from the client to the broker. The *publish* message can be from a client to the broker, or vice versa. It also contains the payload application data. When a client publishes to a topic, the broker will also send a *publish* message with the same payload to any other client who has subscribed to the topic. Finally, the *disconnect* message requests a connection teardown between the client and the broker. Any programming language library that can encode and decode the protocol bytes through a network socket is sufficient to implement the role of a MQTT client. In fact,

we use an open-source implementation [21] to access the functionality of the MQTT client in MicroPython. You can learn about the MQTT protocol in great detail at [22].

5.3. Top module

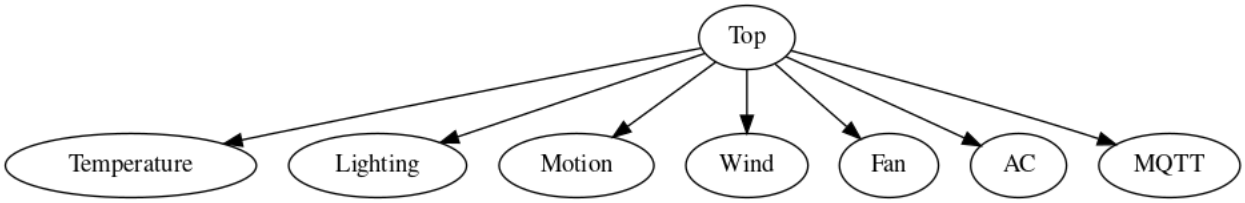

Figure 5.1. The esp32 IoT service module hierarchy

The IoT service has a "*one top module and many submodules*" structure. The top module instantiates each submodule as internal class members. Each submodule handles a specific role, as you can see by Figure 2. Every module also shares a common methods interface. The interface includes the *start*, *stop*, and *routine* functions. Each submodule operates at a specific clock rate. The *routine* function of a submodule is called every clock cycle. Periodic function calling uses the timer scheduled callback API. The timer scheduled callback has two operating modes, which are the one-time mode and the periodic mode. Either mode requires one internal timer hardware module to perform its job. However, there are only four internal timer hardware modules on the esp32 [23]. Therefore, all the submodules have to share one common periodic timer, what we will refer to as a master clock. The master clock has the shortest clock period. Every submodule has a clock period that is a multiple of the master clock period. The three other internal timer hardware modules are reserved for one-time callback uses. We can *init* and *deinit* a timer to start or stop it. *Deinit* can stop the timer before it triggers the callback. The esp32 IoT service can be started by calling the *start* function of the top module. The *start* function of the top module calls *start* functions of submodules to initialize the service. Finally, the *start* function of the top

module *init* the master clock. The *stop* function is similar, but serves to deinitialize resources and *deinit* the master clock. The clock period of the submodules are as follows.

- Master clock: 20 ms
- Temperature: 2020 ms
- Lighting: 380 ms
- Motion: Not applied
- Wind: 20 ms
- Fan: 580 ms
- AC: 620 ms
- MQTT: 460 ms

Notice how the motion submodule does not have a clock rate. This is because the motion submodule operates on a GPIO pin edge-triggered interrupt. Also, notice how the clock periods of the submodules are all prime multiples of the master clock period. This is to avoid having many submodules active in any one master clock cycle. We will denote the term “upstream” to be the data interface between the top module and a submodule, and the term “downstream” to be the data interface between a submodule and the sensor/actuator module it handles. The upstream should eventually reach an MQTT submodule. This module handles sending sensor data and receiving control commands from RPi using the MQTT protocol.

5.4. Temperature submodule

Figure 5.2. DHT22 sensor

The role of the temperature submodule is to get temperature and humidity values from downstream and send the values to the upstream. The submodule clock period is 2020 milliseconds (ms), which is 101 times the master clock period. The temperature and humidity sensor we use has the commercial name of DHT22. This is the only sensor module that does not use a simple data interface. Instead, it uses I2C bus protocol, where data is encoded into protocol bytes for transfer. Temperature and humidity values are available through simple built-in DHT22 MicroPython class. Note that the data reads should be spaced two seconds apart for best accuracy. Still, we choose the number to be 101 as it is the smallest prime number that is larger than 100.

5.5. Lighting submodule

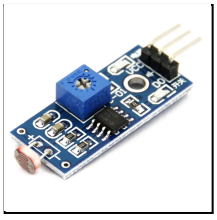

Figure 5.3. CDS - NVZ1 sensor

The role of the lighting submodule is to get active low lighting state from downstream. Then it sends the data inverted to active high to then upstream. The clock period of the

module is 380 ms, which is 19 times the master clock period. The light sensor we use has the commercial name of CDS - NVZ1.

5.6. Motion submodule

Figure 5.4. PIR HC-SR501 sensor

The role of the motion submodule is to get raw data from downstream. Then, it converts the data into binary values representing motion detect status and sends them upstream. The motion sensor we use has the commercial name of PIR HC-SR501. The internal process is as follows. While motion is detected, the downstream sends alternating pulses of three seconds high then three seconds low. If the submodule gets logic 0 from downstream, it calls *init* on the timer class to set the motion status to undetected in five seconds. Otherwise, if the submodule gets logic 1 from downstream, it instead calls *deinit* on the timer and sets the status to detected immediately. Hence, the motion module has a latency of five seconds when switching from the detected to the undetected state.

```
function on_pin_change ( pin, timer ) as
        if pin.value == 1 do
                timer → deinit ( )
                output_one ( )
        else do
                timer → init ( period = 5,  callback = output_zero )
        end
endfunction
```

Algorithm 5.1. Pseudocode for the pin callback function of motion submodule

5.7. Wind submodule

Figure 5.5. YJ-FS RS232/RS285/TTL sensor

The role of the wind submodule is to get raw data from downstream. It then converts them into binary values representing wind detection status to send upstream. The wind sensor module we use has the commercial name of YJ-FS RS232/RS285/TTL. The sensor is similar to the windmill, in that it has 3 blades. The blades rotate when there is wind. At any one angle, the sensor outputs exactly logic 0 or logic 1, deterministically. The 0-giving and 1-giving angles are evenly spaced. Thus the downstream output toggles when the blades rotate. Raw downstream data is thus not in the form we want. We want the wind status to be either logic 1 for detected, or logic 0 for the non-detected state. Also, we only want to catch strong wind signals caused by the fan

and avoid false triggers for when the blades slightly swirl around while the fan is OFF. The way we do it is to count the number of toggles from downstream every clock cycle. If the counter for a cycle is a positive number, we call that cycle *filled*. Otherwise we call the cycle *empty*. Initially, we set the wind status to non-detected. If seventy consecutive cycles are then *filled*, we change the wind status to the detected state. Later, as soon as one cycle reverts back to *empty*, that is enough to change the status back to non-detected. This introduces a latency of seven seconds for a new state to settle. The wind status will also be used as the state of the fan. Lastly, the submodule clock period is 20 ms, equal to the master clock period.

```
function routine ( filled, accum ) as
        if filled is True do
                accum ++
        else do
                accum ← 0
        end
        if accum ≥ 140 do
                output (1)
                accum ← 140
        else do
                output (0)
        end
        filled ← False
endfunction
```

Algorithm 5.2. Pseudocode for the *routine* function of wind submodule

5.8. Fan submodule

The fan submodule gets the fan current state from the wind submodule, instead of the downstream interface. It exposes a *desired* state variable to the upstream interface. It

also maintains an internal busy flag. The submodule does nothing while the busy flag is active. Initially, the busy flag is down. At each clock cycle, the submodule compares the current fan state to the desired fan state. If it is different, the submodule toggles the ON/OFF state of the fan using the IR transmitter (see 5.11). Then, it raises the *busy* flag. Finally, it calls *init* on a timer to clear the busy flag in seven seconds. The seven seconds here is the latency time for a new fan state to settle (see 5.7). After the busy flag is cleared, the processing loop repeats. Note that, while the submodule is busy, the exposed desired state variable is still free to change. That is what enables the user to change the desired fan state at any time so that the desired state is attained as soon as possible. The clock period of the fan submodule is 580 ms, which is 29 times the master clock period. Note that we only choose to work with the ON/OFF command here, which is technically a single IR command. The reason is that basic sensors can only differentiate between the ON and OFF state.

```
function routine ( busy, current, desired, timer ) as
        if busy is False :
                if current is not desired :
                        busy_on ( )
                        toggle_fan_state ( )
                        timer → init ( period = 7, callback = busy_off )
endfunction
```

Algorithm 5.3. Pseudocode for *routine* function of fan submodule

5.9. AC submodule

Figure 5.6. Magnetic door switch sensor

We use a door sensor to detect the state of the AC. The door sensor module we use is a simple magnetic door switch sensor. We basically treat the AC panel the same as a door panel. The door sensor only uses only the ground pin and has no pull-up capability. When the door is closed, the sensor output is shorted to the sensor ground input. When the door is open, the sensor output is left to float. Therefore, we have to enable a weak pull-up resistor for the GPIO pin connected to the sensor output. The data from downstream is already in the ideal active high, binary format, so no further processing is needed. The latency introduced by the door sensor is seven seconds. Such latency is needed for the door to fully reach an open or fully closed state. The AC submodule gets the AC current state from the door sensor, instead of from the downstream interface. It exposes a *desired* state variable to the upstream interface. The submodule also maintains an internal *busy* flag. It does nothing while the busy flag is active. Initially, the busy flag is down. At each clock cycle, the submodule compares the current state to the desired state. If it is different, the submodule toggles the ON/OFF state of AC using the IR transmitter (see 5.11). Then, it raises the busy flag. Finally, it calls *init* on a timer to clear the busy flag in seven seconds. Such is the time needed for a new AC state to settle as mentioned above. After the busy flag is cleared, the processing loop repeats. Note that, while the submodule is busy, the exposed desired state variable is still free to change. That is what enables the user to change the desired AC state at any time so that the desired state is attained as soon as possible. The clock period of the AC submodule is 700 ms. Note that we only choose to work

with the ON/OFF command here, which is technically a single IR command. The reason is that basic sensors can only differentiate between ON and OFF states. Furthermore, AC command codes are complex. The AC remote controller is a stateful electronic device. A button push transmits different command code depending on the current state of the remote controller. Only the ON/OFF button transmits the same command code every time.

```
function routine ( busy, current, desired, timer ) as
        if busy is False :
                if current is not desired :
                        busy_on ( )
                        toggle_AC_state ( )
                        timer → init ( period = 7, callback = busy_off )
endfunction
```

Algorithm 5.4. Pseudocode for *routine* function of AC submodule

5.10. MQTT submodule

The role of the MQTT submodule is to provide an upstream interface for the other submodules to send and receive sensor and control data to and from the RPi controller node. For every other submodule, it provides a class method for them to *publish* their sensor data. The MQTT submodule is also capable of setting the desired state variable that the fan and AC submodule expose. It does so when receiving commands from RPi to change the desired state variable. The *start* function of the submodule first sends the *connect* message to the MQTT broker. Then, it sequentially sends a *subscribe* message to the broker for each control topic it is interested in. Such are the topics of fan-control, AC-control and reset-control. At every clock cycle, the *routine* function would check the mailbox using a non-blocking function call for new commands from RPi to be

carried out. Finally, the *stop* function of the submodule sends the *disconnect* message to the broker to request a connection teardown. Note that the MQTT connection has a keep-alive period. If the broker does not receive any message from a client for as long as the keep-alive period, the client is assumed to be dead. In such a case when connection is suddenly lost, the client can specify a last will message beforehand. The broker will *publish* the last will message of the client when the timeout occurs, so that the RPi controller node is always aware of the current connection status of the esp32. To keep the connection alive, the MQTT submodule has to ping the broker at every clock cycle. Apart from that, The MQTT submodule also has to perform Transport Layer Security (TLS) security procedures when initiating connection with the broker (see 9.2). The submodule clock period is 1100 ms.

5.11. Infrared design

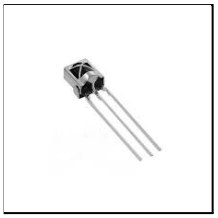

Figure 5.7 & 5.8. TSOP1838 IR receiver and generic IR transmitter LED

IR commands can be sent and received using IR transmitter and receiver LEDs. Our goal is to be able to transmit the ON/OFF command using the transmitter LED. But first, we have to learn the ON/OFF IR signal. We initially use the remote controller that goes with fan and AC to transmit the ON/OFF signal. The receiver LED outputs alternating binary pulses just as it receives the alternating IR signal. We can use GPIO pin edge-triggered callback to capture the IR timing info into a timing array structure. Note that IR signal capturing requires high precision. A slight delay in the edge-triggered callback routine can greatly mess up the timing info. Therefore the

callback routine has to be as short and small as possible. As an example, we should pre-allocate the memory buffer that holds the timing array beforehand to avoid any memory allocation in the callback routine. After capturing, we can decode the timing array using NEC protocol [24] to obtain the IR command code. Note that, although the NEC protocol specifies a unified pulse width length, our fan and AC electronics use their own pulse width length instead. Therefore, we have to approximate these parameters ourselves. There are different pulse widths in an IR signal. We should perform sufficient sampling and save the statistical averages of each pulse width type for later use. With the command code and pulse width info, we are ready to transmit. We can drive the transmitter LED by switching the GPIO pin connected to the LED. However, we cannot do the switching programmatically, it is not precise enough. Thus, we let the built-in RMT class do our switching. The RMT class uses the internal dedicated hardware to switch with extremely high precision. First we need to set the RMT class clock rate according to the pulse width info. We also have to set the RMT modulation frequency to the infrared signal carrier frequency of 38kHz. Finally, we pass the normalized timing array and have RMT perform the switching.

```
function on_pin_change ( pin, last, buffer, index ) as
        now ← timestamp ( )
        if last is not 0 do
                buffer [ index ] ← now - last
                index ++
        end
        last ← now
endfunction
```

```
function routine ( last, index ) as
        now ← timestamp ( )
        diff ← now - last
        if diff > THRES do
                collect_and_decode ( )
                last ← 0
                index ← 0
        end
endfunction
```

Algorithm 5.5. Pseudocode to capture infrared signal

5.12. Routing

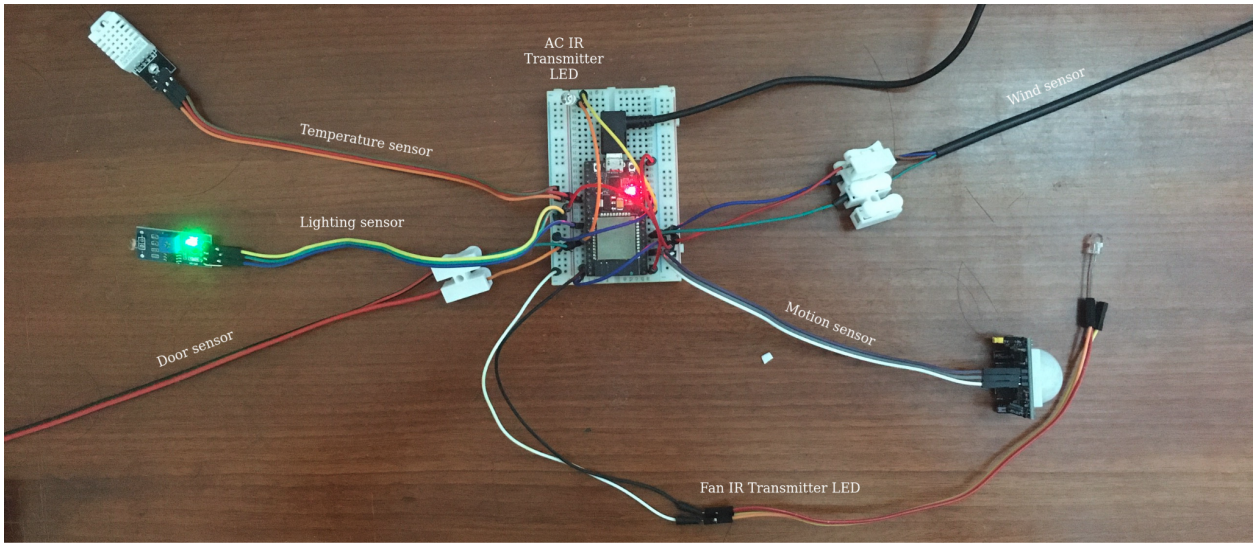

Figure 5.9. Routing schematic

Routing must be done carefully or things could get really convoluted. You can refer to [25] for the pinout of the ESP32-WROOM-32 chip. Note that a power pin connection should be shared between at most 2 modules or esp32 may fail at times. The most common failure is that the wireless interface fails to activate. When that happens, the WLAN class always shows the connection status as disconnected. There are 2 power pins on the esp32, a 5 voltage (V) pin and a 3.3 voltage (V) to be shared among 4 external hardware modules. Typically, 5V gives higher precision than 3.3V. We have to test the precision required of, of each external hardware module and select the two least demanding ones for 3.3V. After careful examination, we decided to connect the motion and wind sensor modules to 5.5V, and leave the temperature and lights sensor module at 3.3V. Note that the door sensor module only needs a ground connection, and the IR transmitter LEDs are directly driven by the GPIO pin. As such, they do not require a power connection.

6. Automation

6.1. Node-red design overview

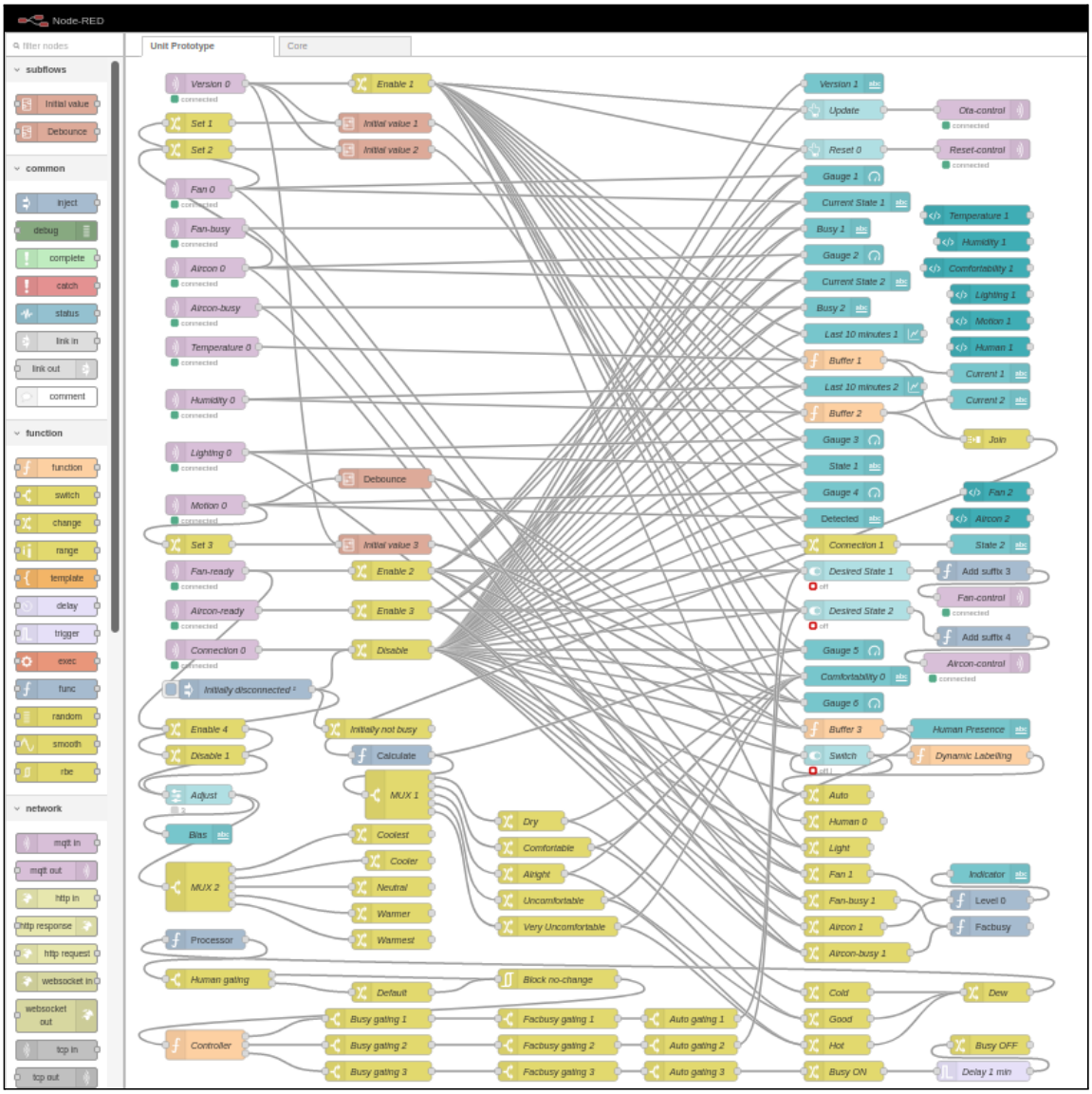

Figure 6.1. The RPi automation controller in a Node-red flow

The esp32 IoT service only provides sensor data and a control interface. It is up to RPi to automate the control. We use the Node-red flow-based visual programming tool to do that. You can refer to [26] for the complete Node-red documentation. A Node-red flow contains nodes. The nodes have inputs, outputs and are linked together. When input arrives, a node performs a function and sends data to its output. Basically, sensor data arrives at MQTT subscriber nodes, triggering the flow. It then traverses through an automatic control rules network to reach MQTT publisher nodes as control commands. Node-red flow-based programming is highly suitable for coding control rules. Normally we program the rules using a lot of nested conditional logic. Instead, we could build a tree of rules in Node-red. Each node in the tree is either a *change* node, or a *switch* node. A change node modifies the input to send to output. A switch node routes input to one of output lines, depending on the input data. This chain of rules structure results in less nested conditional logic and a more transparent and manageable system. Our Node-red flow contains four general classes of nodes as follows.

- MQTT endpoint nodes (the purple nodes in Figure 11)
- Automatic control rules nodes (the bottom yellow portion of Figure 11)
- Enable/disable controller nodes (the yellow nodes labelled Enable/Disable in Figure 11)
- UI nodes (the nuanced-blue nodes in Figure 11)

The enable/disable nodes control when other nodes should be enabled or disabled. Since a node in our system typically requires a pair of enable/disable connections, what results is a lot of wiring that stems from just a few enable/disable nodes. On the other hand, the Node-red UI nodes [27] are used to design a web UI for the end-users. Note that the end-user UI is available at the /ui path relative to the root Node-red URL.

6.2. Heuristic selection

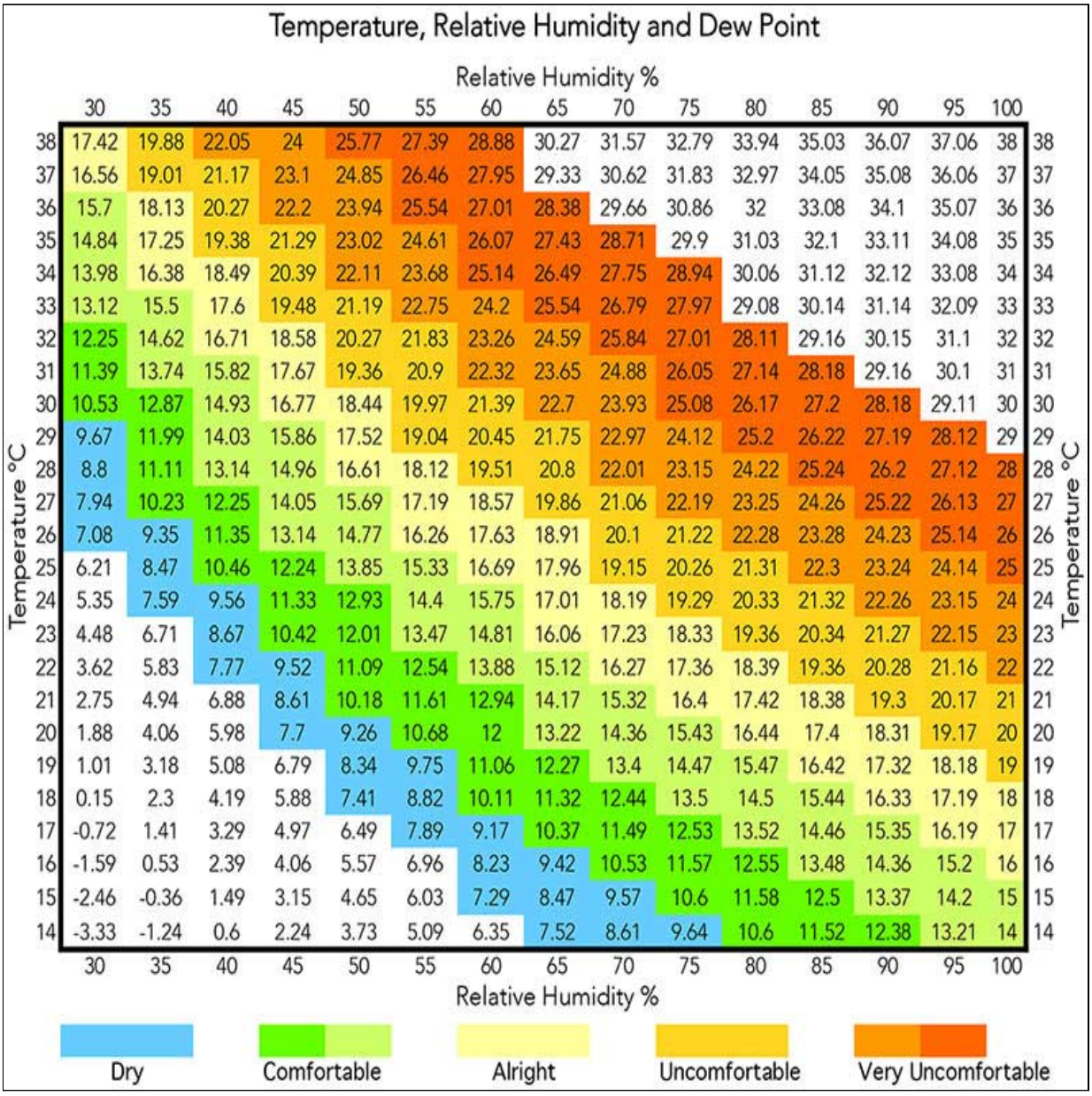

Figure 6.2. Dew point and comfortability chart [28]

Temperature and humidity together affect how comfortable the atmosphere feels. From temperature and humidity we can calculate the dew point temperature using a specific

formula [29]. From the dew point temperature we derive the comfort level heuristic as follows.

- Dew point < 10: Dry
- 10 ≤ Dew point < 16: Comfortable
- 16 ≤ Dew point < 19: Alright
- 19 ≤ Dew point < 22: Uncomfortable
- Dew point ≥ 22: Very uncomfortable

6.3. Automatic control

The fan and AC are always in one out of four possible power states. In descending order of power consumption, they are as follows.

- AC ON and Fan ON
- AC ON and Fan OFF
- AC OFF and Fan ON
- AC OFF and Fan OFF

Next, we will explain the automatic control loop. First in the loop, we check the comfort level heuristic. If it is either *comfortable* or *alright*, we do nothing. If it is *dry* or below, we lower the power state by one level. Otherwise, if it is *uncomfortable* or above, we raise the power state by one level. Then we wait for about 2 minutes for the temperature and humidity sensor to pick up the changes. After that, the loop repeats. There are also additional rules. If there is no human presence detected, we keep both the fan and the AC OFF. Otherwise, if room lights are OFF, we keep the fan OFF and only control the AC. We will have to do a little bit of explanation for the last rule here. We assume that, if lights are OFF, but human presence is detected, that person must be resting. When a person is resting, turning on the fan would be uncomfortable. When resting, people usually prefer a slightly warmer and cozy atmosphere. Furthermore,

turning on the fan would be noisy and consuming power. Also, we let the user adjust their preference for a warmer or cooler atmosphere (see 11.14).

7. Scalability

The system is dynamically scalable in terms of the number of sensor/actuator nodes. One RPi can handle multiple instances of esp32. Each esp32 can be controlled through an end-user UI tab. When a new esp32 enters the system, the end-user UI automatically spawns a new tab. Each UI tab will have the unique machine ID of the esp32 it controls. We will go into details about how to make that work. Node-red can contain multiple flows. One flow should be handling one esp32, which corresponds to a pair of fan and AC. Note that a flow can be exported or imported from Node-red in JSON format. We can use the Node-red HTTP API [30] to retrieve a prototype flow JSON, which is just the flow we hand-crafted earlier to handle only a single esp32 node (see 6.1). Then we edit the flow JSON to parameterize it. Finally, from the parameterized flow JSON, we can use the HTTP API to add a new controller flow, or unit, to Node-red . The text processing steps done on the flow JSON are basically as follows.

- Adding backslash before special escaped characters
- Adding backslash at the end of every line for newline character
- Parameterizing all IDs of the nodes in the flow to avoid ID conflict
- Parameterizing the names of some nodes to identify each esp32 in the web UI

The text processing was done using GNU *sed* and *awk* utilities. What follows is that we get a template flow JSON. Then, in Node-red, we proceed to erase all of the existing flows and create a new core flow. From there on, each time we boot up the RPi service, only the core flow should exist beforehand. It will keep monitoring a dedicated MQTT topic for new esp32 connections. When a new esp32 joins the system, they would send

their machine unique ID to this topic to signal RPi of their arrival. The core flow then uses the template flow JSON generated above to spawn new flows dynamically. Note that adding a new esp32 requires no further configuration, all that has to be done is to simply plug in the power supply. Lastly, scalability of the RPi controller node is beyond the scope of our thesis. We only deal with the scalability of the sensor/actuator nodes, which can scale to any extent assuming there is unlimited processing power and RAM capability on the RPi controller node. Designing for scalability of the controller node would require deep research on many other additional computer science aspects, which includes scheduling, resource management, distributed system, load balancing, advanced network configuration, locational topology, hardware load testing and stress testing, load simulation, etc. Moreover, we would also need to have a lot of microcontroller systems at our disposal.

8. Networking

8.1. LAN configuration

The IoT system has to be functional when the home WiFi router is off. Therefore a LAN has to be provided to esp32 by the Raspberry Pi. Thus, Raspberry Pi will play the role of a wireless access point (AP). Furthermore, Internet connection is also necessary for FOTA update. As such, the LAN has to provide internet capability when the home router is on. Therefore Raspberry Pi will also need to be a wireless client to the home router. We intend to use only the wireless card and no ethernet cables in our setup. The wireless network interface controller (WNIC) will have to switch between the two mentioned roles. Hence the total bandwidth is reduced, and connection may be somewhat slower than normal. That is the reason we also need to be able to switch

back to a pure client network for faster personal uses when desired. Apart from that, The LAN provided by Raspberry Pi also needs to provide a Domain Name System (DNS) service. It is necessary for the Raspberry Pi to have a routable domain name in the LAN. This is to serve the TLS security protocol in the data exchange that happens in the LAN. TLS security is enforced in Node-red HTTP and MQTT protocol exchanges between esp32 and RPi. When connecting, the client would verify the TLS certificate of the server to authenticate them. The subject of certification in our TLS certificates has to be domain names, not private IP addresses. Therefore DNS service is required to give Raspberry Pi a domain name for use in TLS procedure. The TLS protocol will serve to authenticate the RPi server to esp32 clients. That is what we call authentication at the transport level. Meanwhile, the RPi Node-red and MQTT server would authenticate each esp32 client by a username and password tuple. That is what we refer to as authentication at the application level. There is a lack of official or comprehensive guides on how to set up the LAN on RPi. We will have to refer to a variety of online sources. The first is the Raspberry Pi official guide on how to set up RPi as an internet hotspot [31]. However, it requires an ethernet wired connection from RPi to the home router. Fortunately, the second guide is exactly what we need, which is about setting up RPi both as a wireless AP and wireless client [32]. However, it requires us to completely switch from *dchpcd* network manager to *systemd-networkd* network manager first [33]. The setup involves configuring the services of *systemd-networkd*, *hostapd* and *wpa_supplicant* Linux services. However, it does not provide instructions on how to switch back to a pure client network, or how to configure a DNS service. Then there is the third guide, which teaches us how to switch from a *dhcpcd* AP back to a *dhcpcd* wireless client [34], but unfortunately not from a *systemd-networkd* AP. Lucky for us, we did not completely remove *dhcpcd* like instructed in the second guide. Therefore, we can just pick the necessary details from the third guide to help us switch between *systemd-networkd* AP and *dhcpcd* client. The

fourth guide is on how to configure the *dnsmasq* service to set up a DNS server on RPi [35]. Note that we will also have to disable the current DNS resolution service of *systemd-resolved* on RPi first for *dnsmasq* to work. In addition, on other Linux systems that connect to the LAN provided by RPi, we will also have to change the authoritative DNS server to point to the DNS server on RPi. We do so by setting the nameserver attribute of *systemd-resolved* service on those Linux systems to the IP address of RPi.

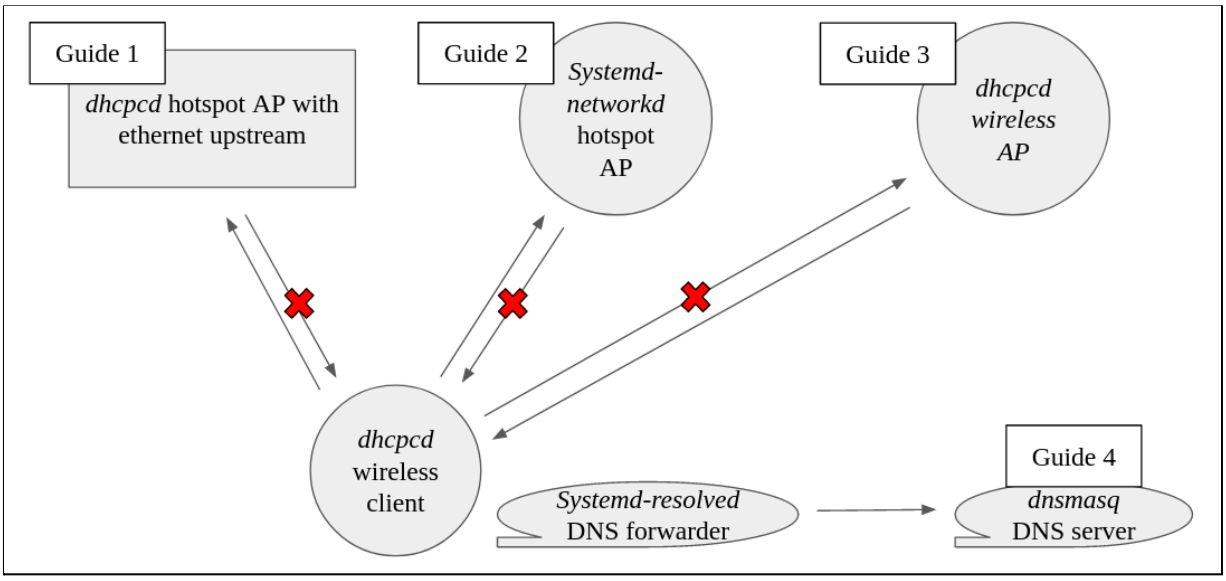

Figure 8.1. Summary of the LAN setup process

8.2. Edge case handling

We have to ensure safe esp32 operation in connection edge cases. We must try-catch every network operation, which includes socket function calls, MQTT function calls and OTA update function calls. Most exceptions occur when the RPi restarts, and the LAN connection is temporarily down. When the LAN connection provided by RPi is lost, the MQTT connection is also lost. When that happens, the esp32 turns off the fan and AC and stops the IoT service. It then waits until the LAN connection goes up again before attempting to reconnect to the MQTT broker and restarting the IoT service. On

the other hand, if the MQTT broker detects that an esp32 MQTT client suddenly disconnects, a last will message of the client will be sent to the last will topic (see 5.2), for the Node-red MQTT endpoints to catch and process accordingly. To sum it up, any node in the system can exit or reconnect at any time and the system would still be functioning correctly, through careful design.

9. Security

9.1. Network Level

The LAN is a private network, hidden behind the home router network address translation (NAT). You cannot normally reach the Node-red or MQTT broker from the wide-area network (WAN) side. However, we still need remote access to the Node-red web interface. We could not use VPN because our home router is hidden behind even more NATs. Knowing your router WAN IP address is a prerequisite to set up the VPN server locally. However, you can still refer to [36] for how to set up a VPN server on RPi. Instead, we have to use the mobile app Remote Red and enable its service on Node-red [37]. Then, we use the mobile app to scan the QR code shown on Node-red. Afterwards, from within the app you can access all of the Node-red HTTP URL paths. This works in a P2P fashion through an initial middle server, possibly through the network hole punching technique [38].

9.2. Transport Level

Inside the local network, both HTTP and MQTT protocol uses TLS. A local CA is maintained as a pair of public and private cryptographic keys. Node-red and the MQTT

broker are given private keys and public certificates signed by the local CA. You can refer to [39] for how to perform the above tasks using the *openssl* linux utility. The CA public certificate is then installed in our web browser and Node-red MQTT client nodes as a trust center in TLS security procedures. Note that, on esp32, the MicroPython secure socket class does not implement CA verification yet. We will have to clone the official MicroPython GitHub repository and merge the pull request that implements CA verification [40] into the main branch build ourselves. Then, we proceed to rebuild and flash the new MicroPython firmware. Also note that CA verification is a very memory-demanding task for microcontrollers with a limited amount of random access memory. The function call will often fail due to out-of-memory exceptions. Therefore, we have to explicitly call garbage collection in MicroPython before and after calling *connect* on the secure socket class.

9.3. Application Level

Node-red and MQTT both do user authentication and authorization. Authentication requires a tuple of username and password from each user or client. Node-red authorization includes restricting read/write permissions on access through the HTTP API or the web UI. On the other hand, the MQTT broker uses an access control file. The file restricts certain users or clients to read/write permissions on certain topics. The file also allows for a dynamic pattern syntax, where a generic user can have permission to a topic whose name is a function of the ID of the user. You can refer to [41] for the access control file syntax.

10. Updatability (FOTA)

10.1. The FOTA flow

Firmware-over-the-air (FOTA) is a subdiscipline of over-the-air (OTA) programming. According to Wikipedia [42], “Over-the-air programming (OTA programming) refers to various methods of distributing new software, configuration settings, and even updating encryption keys to devices like mobile phones, set-top boxes, electric cars or secure voice communication equipment (encrypted 2-way radios). One important feature of OTA is that one central location can send an update to all the users, who are unable to refuse, defeat, or alter that update, and that the update applies immediately to everyone on the channel.” In the FOTA subdiscipline, we are concerned with updating an application through a wireless connection. This gives the ability to update the application on a remote device that is hard to access, or the ability to mass update all nodes within a system, etc. To begin with, we give a brief overview of our FOTA flow. We will refer to the operating code on esp32 as firmware for simplicity, although technically that is not correct. The firmware is put under version control and each version has a version tag. A remote repository (REPO) would store all the firmware versions, each also with a different tag. The version tag of the local firmware is stored in the local file system. Each time esp32 boots up, it would query the remote REPO for the latest version. If the latest version is found to be newer than the local version, the update process starts. In the first step, esp32 queries the remote REPO for a directory file listing (of the latest firmware version). Then, esp32 proceeds to download all the files and folders in the file list to a local temporary folder. After the download is complete, the old firmware is replaced in the file system by the new firmware. Note that the esp32 has to reset and reboot for the update to take effect. During normal operation, when a new firmware version is released, the system admin can

simultaneously reset every esp32 to check for the new update on reboot. For our integration, we pick up an open-source implementation [43] and modify it extensively to suit our needs. Basically, we fix a couple of bugs, change the flow, add exception handling and implement new security functionalities in the library. Now, we will go Into more specific details of the general flow from above. We have the code under version control in a folder named *main* at the root of the local file system. In the *main* folder, a *.version* file stores the local version tag. The remote REPO we mentioned earlier is a Github one. The remote REPO root directory also contains a folder named *main*. The *main* folder is the target of synchronisation between esp32 and the remote REPO. You release a new version on Github by drafting a new version on the current branch head. It lets you attach a version tag and optionally some uploaded assets to the current branch head. Every release can then be referred to using the attached version tag. All communication with Github remote REPO is through the Github HTTP REST API [44]. Back to the esp32, when it boots up, it needs to check the latest remote REPO release. It does so by sending a HTTP GET to Github API server asking for the REPO metadata. The response is a JSON string which contains a latest release version tag field. The esp32 then compares the latest version tag to the local one. If the latest tag is newer, a folder named *next* is created at the local file system root. Inside, the *next* folder, a *.version* file is created to store the latest version tag. The esp32 then sends another request for the remote main folder structure. The response is also a JSON string which has all the files and directories paths relative to the *main* folder. Knowing a file path, the file binary content can be downloaded using another type of request. The returned binary content from the HTTP function call is stored in a MicroPython variable. The variable data is then written to file inside the main folder at the according path. On the other hand, subdirectories of *main* are recursively downloaded the same way as *main*. After the download is complete, the local *main* folder is deleted. The *next* folder is renamed to *main* if supported, otherwise copied to *main* and deleted. We have

to explicitly call the garbage collector before and after every HTTP method call. This is because The HTTP method call may return large data that causes memory exceptions. Such is the case with response JSON strings. Also the maximum raw binary data that can be downloaded and held in RAM is about 45kB. That is to be compared to the total amount of RAM of the esp32, which is 512kB and has to hold the whole MicroPython runtime. Sockets and file streams also need to be closed after each operation to support the tight ram usage. Note that the limited RAM can not whole both the ongoing IoT service and the FOTA update procedure at the same time. That is why the FOTA update process is only available at boot time, before the IoT service is initialized. If a new update is released while the esp32 is performing its IoT service, the user will have the option of mass resetting every esp32 to check for new updates through a special UI tab (see 11.16). That is the whole FOTA flow without any security and safety details. There is a security concern about the authenticity of the downloaded firmware. There is also the safety concern about the behaviour of the update process when the internet is lost mid-download.

```
function fota_update ( repo, preshared_key ) as
        current_version ← content_of ( "/main/.version" )
        latest_version ← get_latest_version( repo, preshared_key )
        if current_version < latest_version do
                mkdir ( "/next" )
                create_new_version_file ( "/next/.version" )
                download_folder_into_next ( repo, shared_secret, "/main" )
                replace_main_by_next ( )
        end
endfunction
```

Algorithm 10.1. Pseudocode for the FOTA update process

10.2. Security

There are two methods to confirm the authenticity of the downloaded file. Note that either method requires the remote REPO to be set under private access beforehand. Thus, ensuring the authenticity of data on the remote REPO can then be left to the Github security. Following that, the first method aims to employ TLS security in a HTTPS exchange. If esp32 can verify the certificate advertised by the server, it will be able to confirm the identity of the server. After that, the connection is guaranteed to be authentic and encrypted, by nature of TLS. The CA verification procedure would be the same as in 9.2. Unfortunately, the CA verification function in our MicroPython build only supports certificates generated from 2048-bit RSA down. Larger certificates are unsupported due to the RAM limitation. Therefore, we have to rely on the second method. The second method verifies the downloaded information and data at the application level by examining the content of them. We can do so by embedding a passphrase in every piece of content we need on the REPO. The example phrase we will use will be *COOKIE*. The downloaded contents include the version tag JSON, the folder file listing JSON and the raw code files. In the metadata JSON string that contains the version tag field, there is also the *assets* subJSON. The *assets* subJSON list files that we attach to a release version, as mentioned in the general flow above. We can attach a file named *COOKIE* to every release version. Then, If the downloaded metadata has an *assets* subJSON that lists a file named *COOKIE*, it is judged to be authentic. Similarly, we put a file named *COOKIE* in the main folder on remote REPO. If the folder file listing JSON does list a file named *COOKIE* then it is authentic. The only issue left is to authenticate the downloaded MicroPython source code text files. If a downloaded MicroPython file has a comment line #COOKIE at the start, then it is authentic. This was convenient as the source code was of text formats. For example, if

it was in binary format with a metadata header, it would have been much more difficult for us to casually add a string to the file binary content.

```
function get_latest_version ( repo, preshared_key ) as
        metadata_json ← get_metadata ( repo )
        if metadata_json [ "assets" ] [ 0 ] [ "name" ] is not preshared_key do
                raise_authenticity_error ( )
        end
        return metadata_json [ "tag_name" ]
endfunction
```

Algorithm 10.2. Pseudocode for secure checking of the latest firmware version

```
function download_folder_into_next ( repo, preshared_key, dir ) as
        directory_listing ← get_directory_listing ( repo )
        secret_file_json ← directory_listing → pop ( )
        if secret_file_json [ "name" ] is not preshared_key do
                raise_authenticity_error ( )
        end
        foreach item in directory_listing do
                if item [ "type" ] is "file" do
                        download_file_into_next ( repo, preshared_key, item [ "path" ] )
                else if item [ "type" ] is "dir" do
                        mkdir ( "next" + dir )
                        subdir ← dir + item [ "path" ]
                        download_folder_into_next ( repo, preshared_key, subdir )
                end
        end
endfunction
```

Algorithm 10.3. Pseudo code for secure downloading of a firmware folder

```
function download_file_into_next ( repo, preshared_key, path ) as
        file_content ← download_file ( repo, path )
        if file_content [ 1 : 9 ] is not preshared_key do
                raise_authenticity_error ( )
        end
        save_file_into_next ( file_content, path )
endfunction
```

Algorithm 10.4. Pseudocode for secure downloading of a firmware file

10.3. Safety

The safety issue is about how we deal with connection lost mid-download. The HTTP GET function will just raise an error if called when the internet is unavailable. However, the function will just poll forever and not raise any error when the internet connection is lost midway. It is because the empty socket can still be read without raising error when that happens. Therefore we have to time them out, and raise an error by program code. After that, we can just catch the errors and process them accordingly. Note that function call timeout in MicroPython is only available in multithreading, multiprocessing modules. We will use *asyncio*, a very safe multithreading library to perform the task. As in most languages, multithreading libraries are a little bit more advanced and complicated than the typical use cases. You can refer to the details of the MicroPython *asyncio* module from its CPython counterpart guidebook [45].

11. Web UI

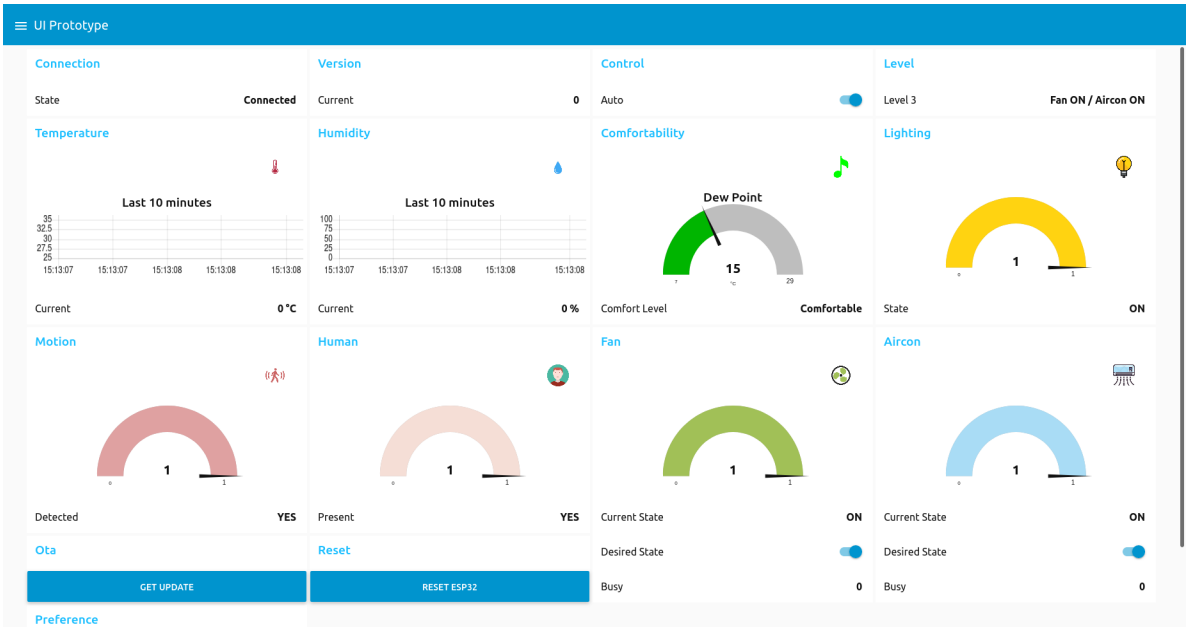

Figure 11.1. The Web UI when most components are turned on

11.1. Tab menu

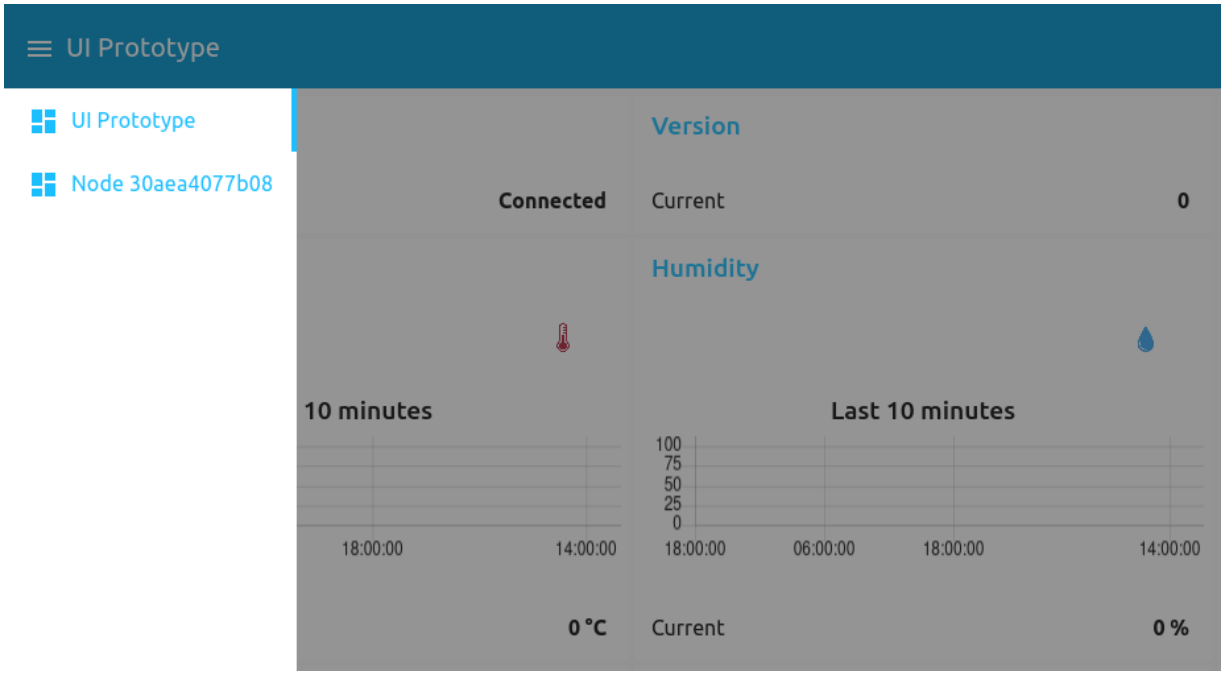

Figure 11.2. UI Tab menu

A dropdown tab menu appears when you click the top left UI icon. The tab menu lets you switch between different esp32 nodes. The name of a UI tab corresponds to the machine unique ID of the esp32 it handles. There is also a special UI tab that lets you mass update every esp32 (see 11.16).

11.2. Connection

Figure 11.3. UI *Connection* panel

The *Connection* panel shows the connection status between RPi and esp32. *State* is either *Connected* or *Disconnected.* When the *state* is *Disconnected*, the rest of the UI is disabled.

11.3. Version

Figure 11.4. UI *Version* panel

The *Version* panel shows the current version of the esp32 IoT service. The version number is in dot-decimal notation. Typically, when we release a new firmware version, we will increment the version number by one unit of the last digit. The digits will be of base 10 and will carry over to the left as they increment from 9 back to 0.

11.4. Control

Figure 11.5. UI *Control* panel

The *Control* panel has a switch to toggle between *Manual* and *Automatic* mode. The *Manual* mode lets you access the *Fan*, *Aircon* and *Preference* and Reset panels. Meanwhile, the *Automatic* mode only lets you access the *Reset* panel.

11.5. Level

Figure 11.6. UI *Level* panel

The *Level* panel shows the current fan and AC power state. The 4 possible power state levels are as follows.

- Level 0: Fan OFF / AC OFF
- Level 1: Fan ON / AC OFF
- Level 2: Fan OFF / AC ON
- Level 3: Fan ON / AC ON

11.6. Temperature

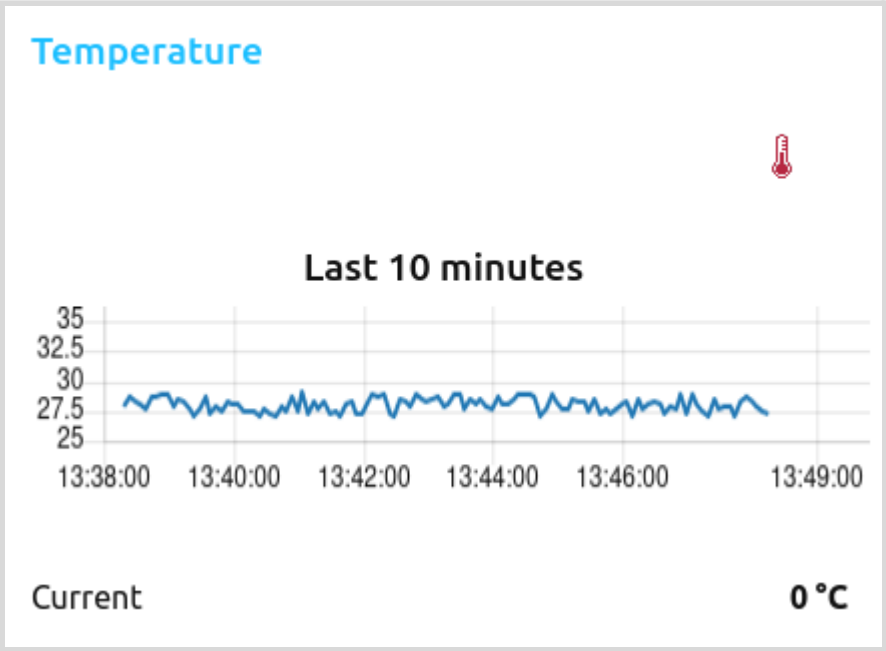

Figure 11.7. UI *Temperature* panel

The *Temperature* panel shows the current temperature value. It also has a temperature chart of the last ten minutes. The value is represented in Celsius degrees (°C). The chart will only show temperature points between 25°C to 35°C, as the measure value typically stays within these boundaries.

11.7. Humidity

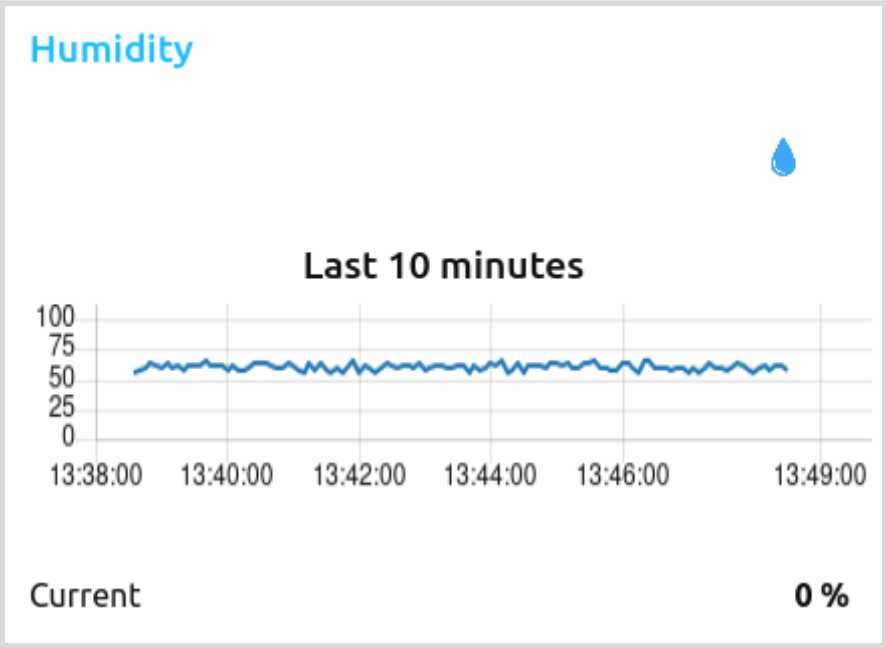

Figure 11.8. UI *Humidity* panel

The *Humidity* panel shows the current humidity value. It also has a humidity chart of the last ten minutes. The humidity measure is represented as relative humidity, in units of percentage (%).

11.8. Comfortability

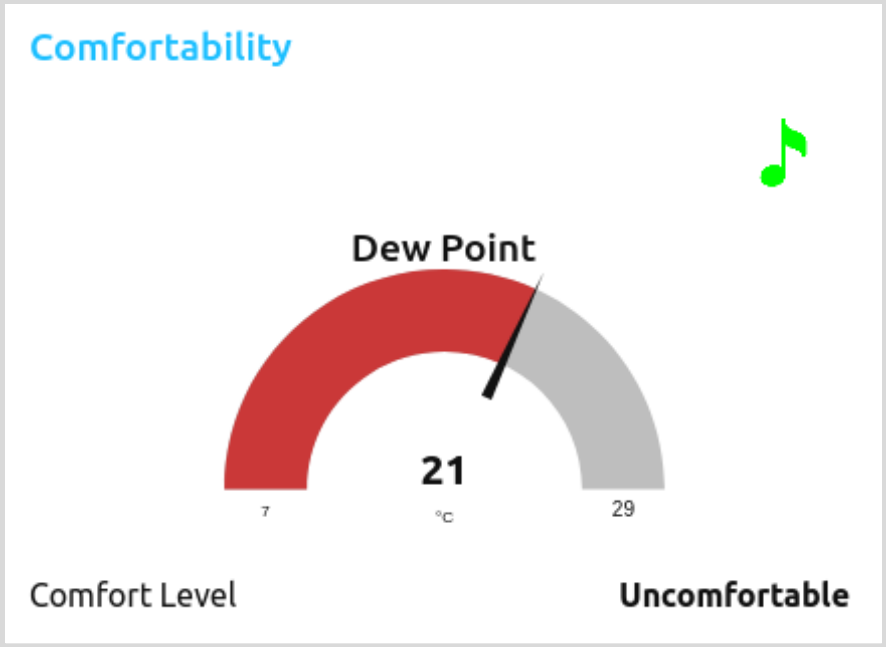

Figure 11.9. UI *Comfortability* panel

The *Comfortability* panel has a gauge that shows the current dew point temperature. Dew point temperature is calculated from temperature and humidity using a specific formula. *Comfort Level* is the heuristic depending on the dew point temperature. The possible levels and their conditions are as follows.

- Dew point < 10: Dry
- 10 ≤ Dew point < 16: Comfortable
- 16 ≤ Dew point < 19: Alright
- 19 ≤ Dew point < 22: Uncomfortable
- Dew point ≥ 22: Very uncomfortable

11.9. Lighting

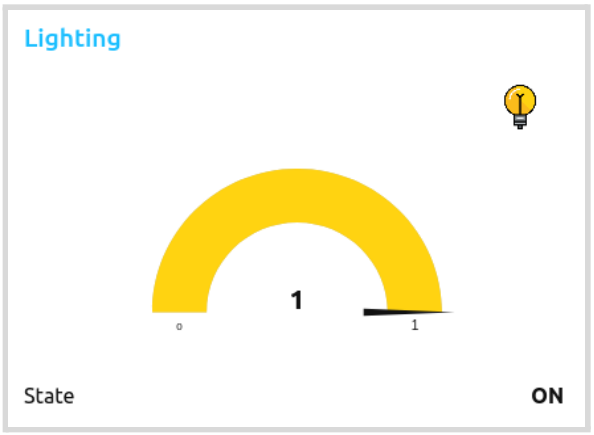

Figure 11.10. UI *Lighting* panel

The lighting state is either ON or OFF. There is also a gauge in yellow to provide visual color aid.

11.10. Motion

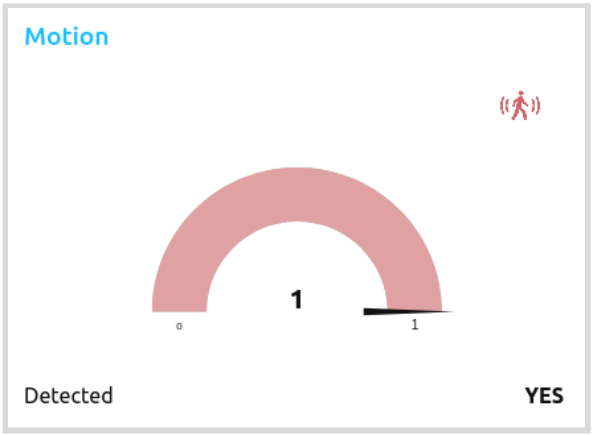

Figure 11.11. UI *Motion* panel

The motion detected status is either YES or NO. There is also a gauge in red to provide visual color aid.

11.11. Human

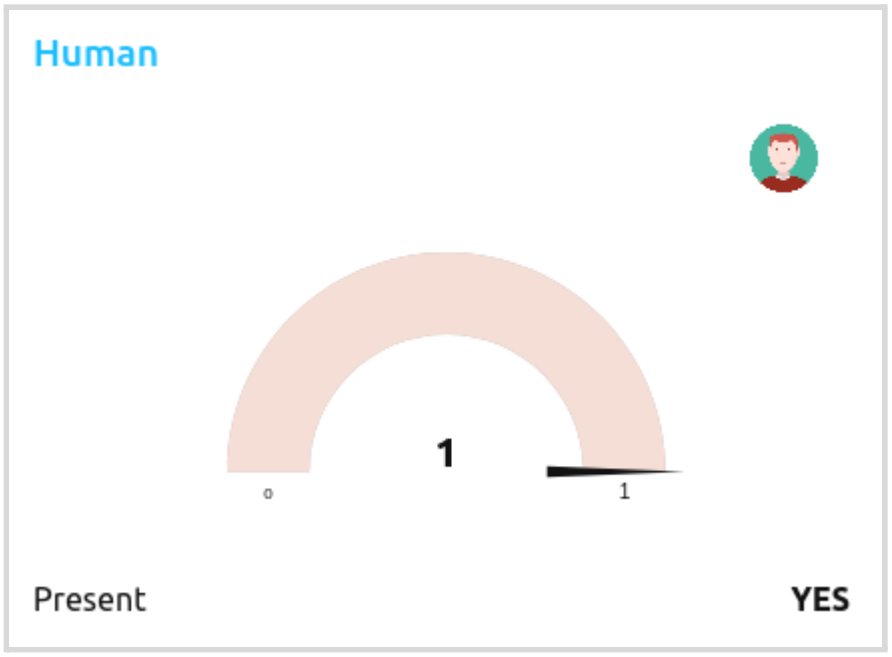

Figure 11.12. UI *Human* panel

The human present status is either YES or NO. Note that a new motion state has to hold for at least 2 minutes for the *Human Present* state to update. There is also a gauge in light skin to provide visual color aid.

11.12. Fan

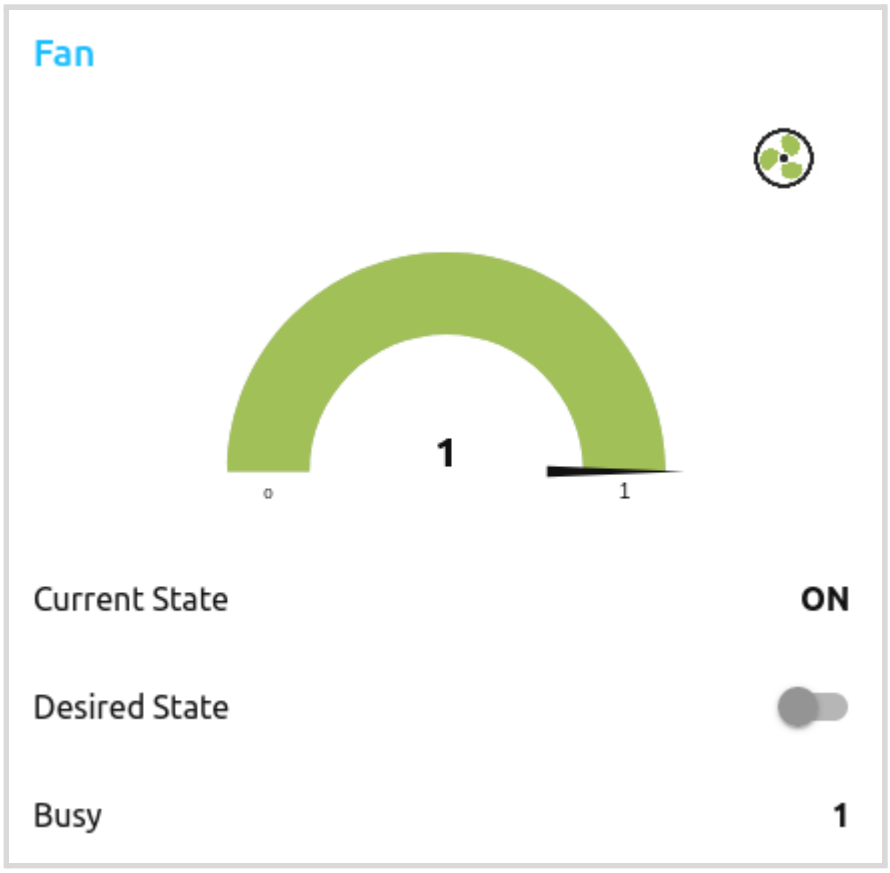

Figure 11.13. UI *Fan* panel

Current State is either ON or OFF. *Desired State* provides a switch to set the desired fan state. Initially the *Desired State* is the same as *Current State.* The esp32 always tries to keep the first fan state it detected when it starts. The *Busy* flag is either 0 or 1. The esp32 raises the *Busy* flag if it is waiting for a new fan state to settle in. After the *Current State* finally reflects the *Desired State*, the *Busy* flag is dropped. The *Desire State* slider is disabled in *Automatic* mode.

11.13. Aircon

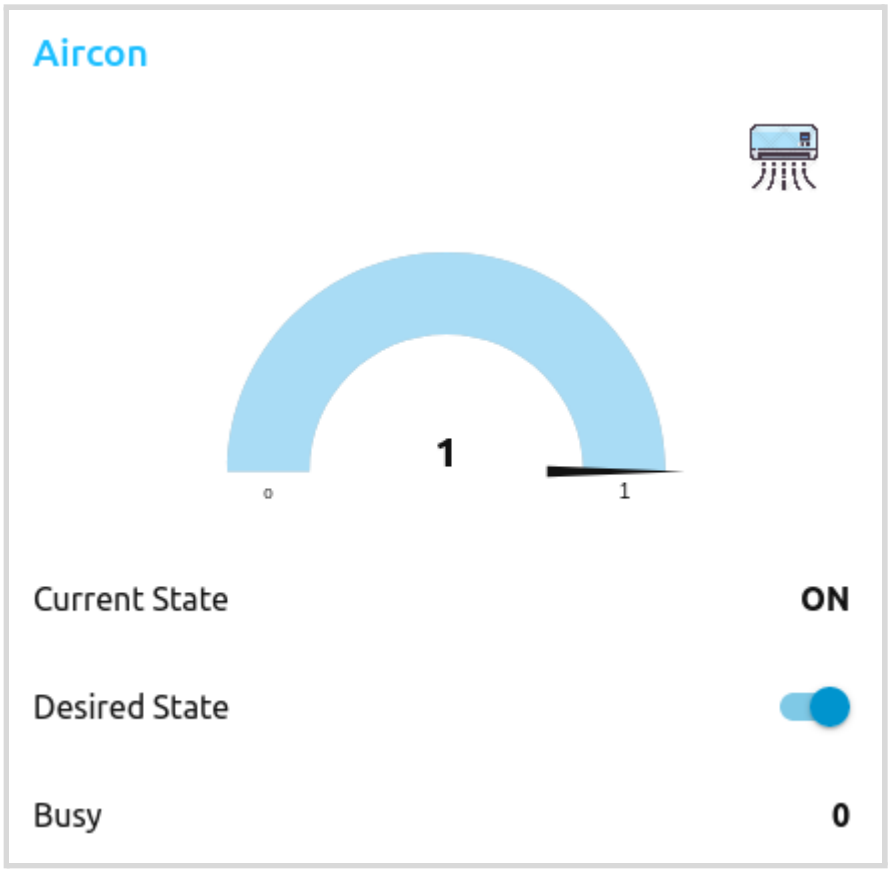

Figure 11.14. UI *Aircon* panel

Current State is either ON or OFF. *Desired State* provides a switch to set the desired AC state. Initially the *Desired State* is the same as *Current State*. The esp32 tries to keep the first AC state it detected when it starts. The *Busy* flag is either 0 or 1. The esp32 raises the *Busy* flag if it is waiting for a new AC state to settle in. After the *Current State* finally reflects the *Desired State*, the *Busy* flag is dropped. The *Desire State* slider is disabled in *Automatic* mode.

11.14. Preference

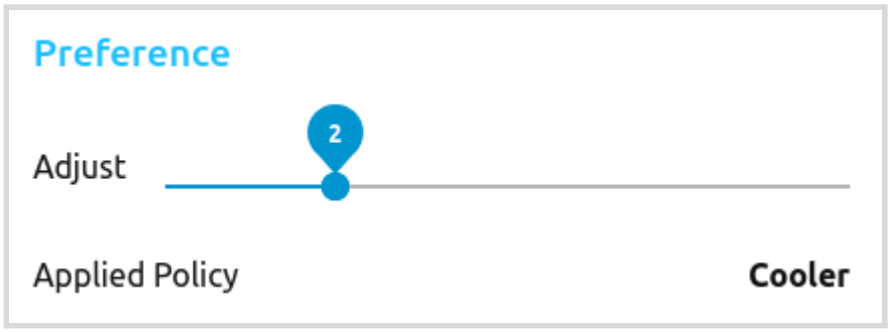

Figure 11.15. *Preference* panel

The *Preference* panel has a slider to let the end-user adjust their preference. There are five applicable preferences. From the left to the right end of the slider, they are *Coolest*, *Cooler*, *Neutral*, *Warmer*, *Warmest.* The choice here affects the calculated dew point temperature as follows. The *Cooler* and *Coolest* policy add +1.5 and +3 to the dew point value correspondingly. The *Warmer* and *Warmest* policy add -1.5 and -3 to the dew point value correspondingly. The default policy *Neutral* does not modify the dew point value. The panel is disabled in *Automatic* mode.

11.15. Reset

Figure 11.16. *Reset* panel

The *RESET* button commands esp32 to perform a reset. This tries to reset esp32 immediately without turning off the fan or AC.

11.16. FOTA Update

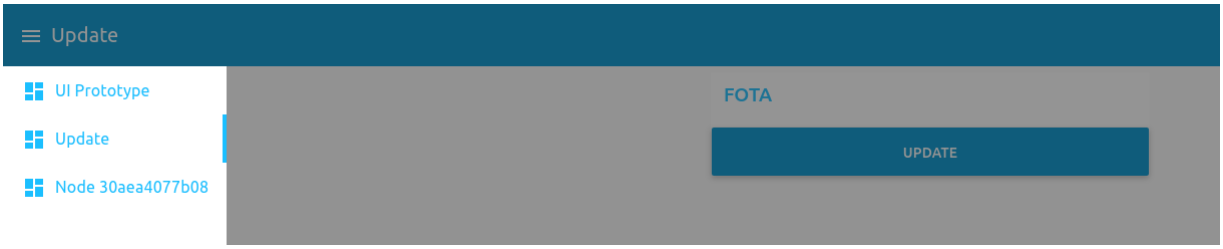

Figure 11.17. *Update* UI Tab and *FOTA* menu

There is a special and separate *Update* UI tab which handles mass deploying updates to every esp32 device. By clicking *UPDATE* button on the *FOTA* menu, the user commands every esp32 to immediately reset and check for a new firmware version update on reboot. Note that a new update can only be checked and installed at boot time. This is due to memory limitations. The whole RAM can not whole both the ongoing IoT service and the FOTA update procedure at the same time. Only at boot time, when the IoT service is yet to be initialized, that it is possible to perform the FOTA update process without raising memory exceptions. You can refer to 10.1 for specific details.

12. Result

We successfully built a home automation IoT application to refine the indoor atmosphere by controlling the fan and AC, the details of which are as follows. A web UI is provided to the end user to control the application. Each tab in the web UI represents one esp32 device that controls a pair of fan and AC. A new esp32 can be added to the system for each pair of fan and AC without any further configuration needed. The web UI will automatically spawn a new tab when an esp32 goes online.

Through the UI, the user can monitor the temperature, humidity, light, human presence, fan and AC states. The user can also set the desired state for fan and AC in manual mode, or let the system set the fan and AC state on its own in automatic mode. In the automatic mode, the system tries to keep the room atmosphere comfortable, one which is not too hot or cold. It also tries to conserve fan and AC power as much as possible while the air is kept pleasant. In manual mode, when setting the desired fan AC state, the user should wait a little bit for the new state to settle in on the web UI. The user can expect a new fan and AC state to be reflected in seven seconds. At any time, the user can click the reset button to reset an esp32, or command every esp32 to check for a new update simultaneously through a special UI menu. Versions of the system are stored on a remote GitHub repository. Update can be downloaded and installed from a repository every time the esp32 boots. The update process is guaranteed to be secure using an ad-hoc application-level mechanism. The system is hidden in a private network, although remote access is still possible through a specific mobile app. Inside the local network, communication between the nodes is also guaranteed to be secure under TLS protection. Lastly, when connection is lost, the esp32 should know to turn off both fan and AC to save energy.

13. Limitations

13.1. IoT limitations

To begin with, we only work with the simple ON/OFF fan and AC commands. The reason is that basic sensors can only differentiate between ON and OFF states. Furthermore, AC command codes are complex. The AC remote controller is a stateful electronic device. A button push transmits different command code depending on the

current state of the remote controller. Only the ON/OFF button transmits the same command code every time.

13.2. Automation rules limitations

The automatic control rules we employ are still quite simple. It does not involve any machine-learning yet.

13.3. Scalability limitations

We only design for the dynamic scalability of the esp32 sensor/actuator nodes. Design for the scalability of the Ri controller node is beyond the scope of our thesis. You can refer to all the details at 5.

13.4. Remote access limitations

Remote access is only available through the limited mobile screen. We could not use VPN because our home router is hidden behind more NATs.

13.5. Updatability (FOTA) limitations

The maximum size of a MicroPython source code file that can be downloaded when performing FOTA update is about 45 KB. A downloaded file has to be stored in RAM first to verify its authenticity before being written to the file system. This is because we authenticate a downloaded source code text file by the content of it. We could not have TLS verify the authentication of the server at the transport level, due to memory limitation. If the downloaded file is instead written directly to persistent storage without buffering in RAM first, the maximum file size that can be downloaded would be at least 1 MB. Also for the reason of memory limitations, esp32 can only perform

FOTA updates at boot, before the IoT service is initialized. The RAM does not have sufficient capacity to host the IoT service and perform updates at the same time. Thus, during normal operation, the owner has to force every esp32 to reset to check for and install a new update. You can refer to all details at 10.1 and 10.2.

13.6. Usability limitations

We have to differentiate between different esp32 by their not readily memorable unique machine IDs. We also have to attach the ID label on each physical esp32 board.

13.7. Other limitations

The LAN setup on RPi (see 8.1) has some side effects. The setup will make us unable to *sshfs* the RPi file system on another PC anymore. Instead, we can only use *sshfs* to mount another PC file system on the RPi. Also the taskbar network icon on RPi is forever corrupted after the setup. This is due to the fact that, to make the whole setup work, we had to mix steps from different guides that, when applied together, could have unseen effects.

14. Conclusion

To sum up, we propose an IoT system to control household fans and ACs. We seek to tackle all the common IoT challenges of security, scalability, updatability, networking and automation. This paper describes in detail the implementation process and the problems we came across. It could serve as reference for anyone that is searching for a partial or full-flow IoT solution. Lastly, we hope to promote infrared application as a non-intrusive home renovation process.

15. Future work

We will mainly focus on addressing the two limitations we discussed at 13.1 and 13.2. Regarding the first, we will try to incorporate more complex infrared commands into our application. We could use more advanced types of sensors to differentiate between complex fan or AC states. We could also put efforts into fully emulating the AC remote controller finite state machine. Regarding the second, we will look into more formal methodologies that infer comfort level and controls from input sensor data to upgrade our basic automatic control scheme. Additionally, we could begin to introduce machine-learning techniques into our proposed model.

References

[1] V. D. Vaidya and P. Vishwakarma, “A Comparative Analysis on Smart Home System to Control, Monitor and Secure Home, based on technologies like GSM, IOT, Bluetooth and PIC Microcontroller with ZigBee Modulation,” in Int. Conf. on Smart City and Emerg. Technol. (ICSCET), Mumbai, India, 2018, doi: 10.1109/ICSCET.2018.8537381.

[2] T. K. Gannavaram V, U. M. Kandhikonda, R. Bejgam, S. B. Keshipeddi and S. Sunkari, “A Brief Review on Internet of Things (IoT),” in Int. Conf. on Comput. Commun. and Inform. (ICCCI), Coimbatore, India, 2021, doi: 10.1109/ICCCI50826.2021.9451163.

[3] Q. Qi; X. Xie; C. Peng and J. Hu, “Design of Wireless Smart Home Safety System Based on Visual Identity,” in Int. Conf. on Commun. Inform. Syst. and Comput. Eng. (CISCE), Beijing, China, 2021, doi: 10.1109/CISCE52179.2021.9445989.

[4] S. S. I. Samuel, “A review of connectivity challenges in IoT-smart home,” in 3rd MEC Int. Conf. on Big Data and Smart City (ICBDSC), Muscat, Oman, 2016, doi: 10.1109/ICBDSC.2016.7460395.

[5] P. Kar and H. Wang, “EZPlugIn: Plug-n-Play Framework for a Heterogeneous IoT Infrastructure for Smart Home,” in IEEE Internet of Things Magazine, 2021, doi: 10.1109/IOTM.0001.2000172.

[6] B. Mishra and A. Kertesz, “The Use of MQTT in M2M and IoT Systems: A Survey,” in IEEE Access vol. 7, 2020, doi: 10.1109/ACCESS.2020.3035849.

[7] B. Mustafa, M. W. Iqbal, M. Saeed, A. R. Shafqat, H. Sajjad and M. R. Naqvi, “IOT Based Low-Cost Smart Home Automation System,” in 3rd Int. Congr. on Human-Comput. Interact., Optim. and Robot. Appl. (HORA), Ankara, Turkey, 2021, doi: 10.1109/HORA52670.2021.9461276.

[8] K. Agarwal, A. Agarwal and G. Misra, “Review and Performance Analysis on Wireless Smart Home and Home Automation using IoT,” in Third Int. conf. on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Palladam, India, 2019, doi: 10.1109/I-SMAC47947.2019.9032629.

[9] A. Khan, A. Al-Zahrani, S. Al-Harbi, S. Al-Nashri and I. A. Khan, “Design of an IoT smart home system,” in 15th Learn. and Technol. Conf. (L&T), Jeddah, Saudi Arabia, 2018, pp. doi: 10.1109/LT.2018.8368484.

[10] F. H. Purwanto, E. Utami and E. Pramono, “Design of server room temperature and humidity control system using fuzzy logic based on microcontroller,” in Int. Conf. on Inform. and Commun.Technol. (ICOIACT), Yogyakarta, Indonesia, 2018, doi: 10.1109/ICOIACT.2018.8350770.

[11] B. Ali, “Internet of Things based Smart Homes: Security Risk Assessment and Recommendations” M.S. thesis, Dept. Comput. Sci., Elect. and Space Eng., Luleå Univ. of Technol., Luleå, Sweden, 2016.

[12] A. Yohan; N. Lo and L. P. Santoso, “Secure and Lightweight Firmware Update Framework for IoT Environment,” in IEEE 8th Global Conf. on Consum. Elect. (GCCE), Osaka, Japan , year, doi: 10.1109/GCCE46687.2019.9015316.

[13] M. Bettayeb, Q. Nasir and M. A. Tablib, “Firmware Update Attacks and Security for IoT Devices,” in ArabWIC 6th Annu. Int. Conf., Rabat, Morocco, 2019, doi: 10.1145/3333165.3333169.

[14] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig and E. Baccelli, “Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check,” in IEEE Access vol. 7, 2019, pp. 71907 - 71920, doi: 10.1109/ACCESS.2019.2919760.

[15] A. Luntovskyy, L. Globa, “Performance, Reliability and Scalability for IoT,” in Int. Conf. on Inform. and Digit. Technol. (IDT), Zilina, Slovakia, 2019, doi: 10.1109/DT.2019.8813679.

[16] T. Adiono, B. A. Manangkalangi, R. Muttaqin, S. Harimurti and W. Adijarto, “Intelligent and secured software application for IoT based smart home,” in IEEE 6th Global Conf. on Consum. Elect.(GCCE), Nagoya, Japan, 2017, doi: 10.1109/GCCE.2017.8229409.

[17] I. Ullah and Q. H. Mahmoud, “Network Traffic Flow Based Machine Learning Technique for IoT Device Identification” in IEEE Int. Syst. Conf. (SysCon), Vancouver, Canada, 2021, doi: 10.1109/SysCon48628.2021.9447099.

[18] T. Perumal; Y. L. Chui; M. A. B. Ahmadon; S. Yamaguchi, “IoT based activity recognition among smart home residents,” in IEEE 6th Global Conf. on Consum. Elect. (GCCE), Nagoya, Japan, 2017, doi: 17451933.

[19] *MQTT*. en.wikipedia.org. https://en.wikipedia.org/wiki/MQTT (accessed Jul. 15, 2021).

[20] D. George. “MicroPython documentation.” docs.MicroPython.org. https://docs.MicroPython.org/en/latest/ (accessed Jul. 15, 2021).

[21] R. Santos and S. Santos, “MQTT Protocol,” in *MicroPython Programming with ESP32 and ESP8266 v1.2*. Porto, Portugal: R. Santos and S. Santos, ch. 5, sec. 3, pp. 231–245.

[22] *MQTT & MQTT 5 Essentials*. HiveMQ GmbH, 2020. [Online]. Available: https://www.hivemq.com/downloads/hivemq-ebook-mqtt-essentials.pdf. Accessed: Jul. 15, 2021.

[23] Espressif Systems Co., Ltd., Shanghai, China. *ESP32 Series Datasheet v3.6*. (2021). Accessed Jul. 15, 2021. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[24] S. Bergmans. “NEC Protocol.” exploreembedded.com. https://exploreembedded.com/wiki/NEC_IR_Remote_Control_Interface_with_8051 (accessed Jul. 15, 2021)

[25] Espressif Systems Co., Ltd., Shanghai, China. *ESP32-WROOM-32 Datasheet v3.1*. (2021). Accessed Jul. 15, 2021. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf

[26] Node-red documentation. nodered.org. https://nodered.org/docs/ (accessed Jul. 15, 2021).

[27] D. Jones, N. O'Leary and D. Cunnington. “node-red-dashboard.” flows.nodered.org. https://flows.nodered.org/node/node-red-dashboard (access Jul. 15, 2021).

[28] P. Wilson. “Climate Control.” mrfixitbali.com. https://www.mrfixitbali.com/air-conditioning/temperature-humidity-dew-point-251.html (accessed Jul. 15, 2021).

[29] *Dew point*. en.wikipedia.org. https://en.wikipedia.org/wiki/Dew_point#Calculating_the_dew_point (accessed Jul. 15, 2021).

[30] Node-red HTTP API Documentation. nodered.org. https://nodered.org/docs/api/admin/ (access Jul. 15, 2021).

[31] Setting up a Raspberry Pi as a routed wireless access point. raspberrypi.org. https://www.raspberrypi.org/documentation/configuration/wireless/access-point-routed.md (accessed Jul. 15, 2021).

[32] Ingo. Wireless access point and client connection on a single interface wlan0. raspberrypi.stackexchange.com. https://raspberrypi.stackexchange.com/a/93636 (accessed Jul. 15, 2021).

[33] Ingo. Setting up systemd-networkd. raspberrypi.stackexchange.com. https://raspberrypi.stackexchange.com/a/108593 (accessed Jul 15, 2021).

[34] Hydrosys4. “RPI3 switch between wifi AP and client”. hydrosysblog.wordpress.com. https://hydrosysblog.wordpress.com/2016/08/07/rpi3-switch-between-wifi-ap-and-client/ (Accessed Jul. 15, 2021).

[35] A. Kili. “How to Setup a DNS/DHCP Server Using dnsmasq on CentOS/RHEL 8/7”. tecmint.com. https://www.tecmint.com/setup-a-dns-dhcp-server-using-dnsmasq-on-centos-rhel/ (accessed Jul. 15, 2021).

[36] N. Le. VPN server setup on Raspberry Pi. codelearn.io (in Vietnamese). https://codelearn.io/sharing/tao-vpn-server-voi-raspberry-pi (accessed Jul. 15, 2021).

[37] Setting up Remote-red. remote-red.com. https://www.remote-red.com/en/help/ (accessed Jul. 15, 2021).

[38] *Hole punching (networking)*. en.wikipedia.org. https://en.wikipedia.org/wiki/Hole_punching_(networking) (accessed Jul. 15, 2021).

[39] M. Anicas. “OpenSSL Essentials: Working with SSL Certificates, Private Keys and CSRs”. digitalocean.com. https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs (accessed Jul. 15, 2021).

[40] Pumelo. “esp32: Add ca_cert to ussl”. github.com. https://github.com/MicroPython/MicroPython/pull/5998 (accessed Jul. 15, 2021).

[41] Steve. “Mosquitto ACL -Configuring and Testing MQTT Topic Restrictions”. steves-internet-guide.com. http://www.steves-internet-guide.com/topic-restriction-mosquitto-configuration/ (accessed Jul. 15, 2021).

[42] *Over-the-air programming*. en.wikipedia.org. https://en.wikipedia.org/wiki/Over-the-air_programming (accessed Jul. 15, 2021).

[43] R. Dehuysser. “MicroPython OTA Updater.” github.com. https://github.com/rdehuyss/MicroPython-ota-updater (accessed Jul. 15, 2021).

[44] GitHub REST API *Repositories* reference. docs.github.com. https://docs.github.com/en/rest/reference/repos (accessed Jul. 15, 2021).

[45] C. Hattingh, *Using Asyncio in Python: Understanding Python's Asynchronous Programming Features*, 1st ed. Sebastopol, CA, USA: O’Reilly Media, Inc., 2020.

Appendix

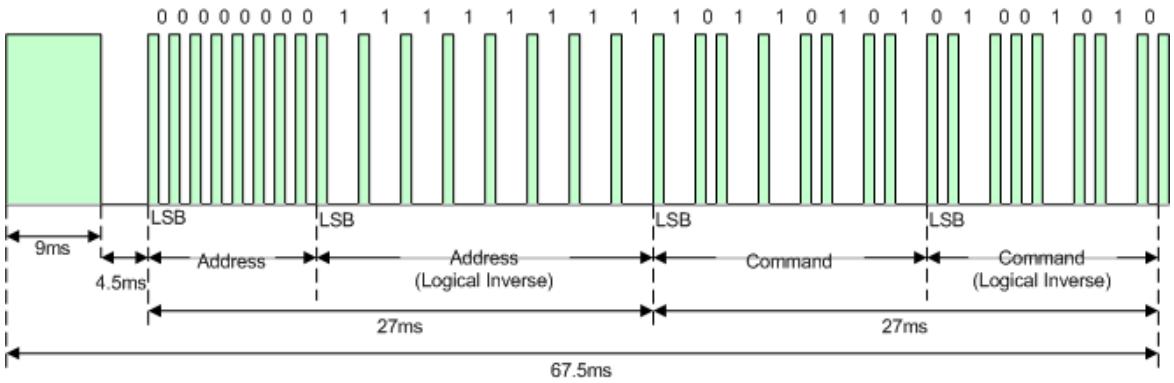

Figure 1. Example message frame using the NEC IR transmission protocol