HO CHI MINH CITY, UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEER

Application-Based Internet of Things Report

Student: Nguyen Dinh Minh

HO CHI MINH CITY, 12/2021

Content

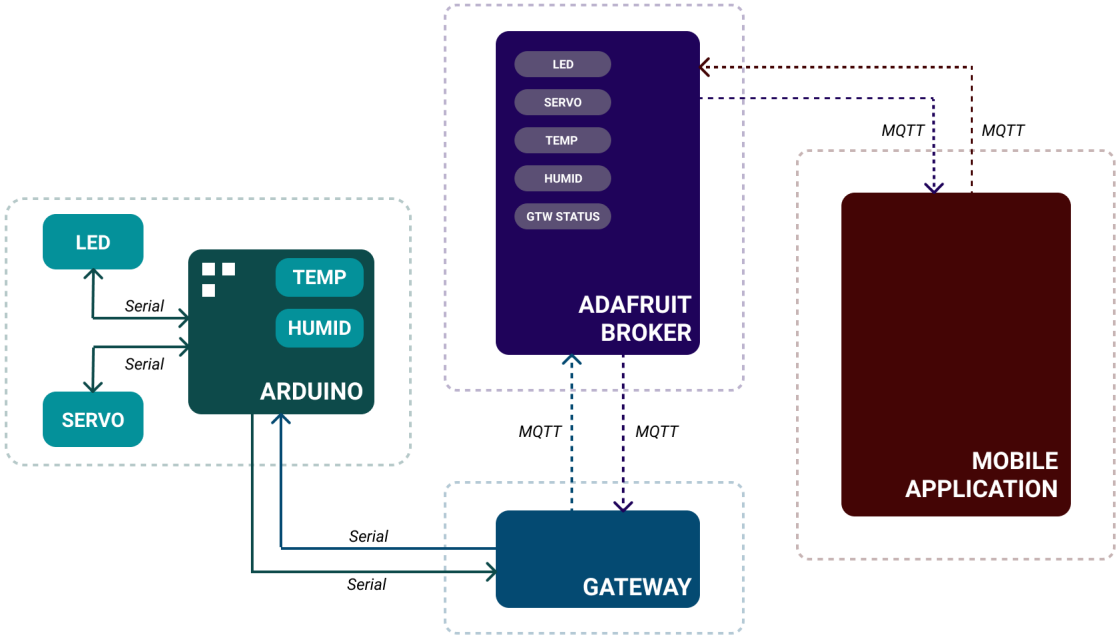

Figure 1: IoT System Layers

1 Introduction

The application-based IoT system used in this project involves 4 main layers of the standard IoT environment. The abstract view of the system is depicted in Figure 1, in which the 4 main modules are highlighted in 4 different colors and the communication protocols between them are also included as interconnecting arrows. Communication involves both wireless protocols (depicted as dash arrows) and wired connections (depicted as normal arrows). The names of the communication protocol being used are typed right beside the corresponding arrows. We only have 2 communication protocol in this project: MQTT protocol and UART communication. The implementation of these protocols will further be explain in more detail later.

The first module of our system is the **Arduino Node**. This layer of the IoT system do the simple assembly work that receives data from the external *Led* and *Servo* device, as well as generates random *Temperature* and *Humidity* value. The *Arduino* is also responsible for sending these data packets through UART communication to the next module, the Gateway layer.

Acting as an intermediate module, the **Gateway** constantly monitors the incoming data packets from both the Arduino and the Adafruit Broker. Its main job is to handle these data and dispatch (send) them to the correct destination. An additional and important feature of the Gateway is to fetch the latest data from the broker and send them to the Arduino node when it first connect (or reconnect) to the server. The implementation of the Gateway will also be explained in more detail in later part of the document.

The broker used in this project is the service provided by the **Adafruit IO** cloud. The broker provides enough API implementation for MQTTv3.1 standard. Our IoT system will have 5

separate feeds on the server to record and distribute data to any client connecting to the broker. We have the *Gateway* is the first MQTT client by which we publish data packets from the *Arduino Node* to the online feeds. Gateway client also subscribes to several (but not all) necessary topics from the broker to receive request messages from the server. The broker maintains an additional feed called `gtw-status` that record the current status (*online*/*offline*) of the Gateway. Whenever our Gateway successfully connects to the broker, it will automatically set its status to *online* by publish a short message to this feed. In the event the Gateway is **willing** to disconnect from the server, it will publish an *offline* message to the status topic before leaving. This will be further explained when we come to the implementation of the Gateway.

The second MQTT client, also the last layer of out IoT system, is the **mobile application** module. It is at this layer the users have a decent interface to monitor and interact with the IoT components. Much like the Gateway, our mobile application implements the v3.1 version of the MQTT protocol to communicate with the Adafruit server. The detail of the mobile layer is saved until we come to the section 5.

Now that's the overview to the IoT architecture. We now continue to explore how to actually put these modules down in the upcoming sections. Before going on, in order to ease up tracking the implementation, the whole project can be found at this Github repository hosted by the author of this document.

2 Arduino Implementation

All the sensor nodes are boiled down to the Arduino. We have 2 actual sensor simulators: the **built-in Arduino's led** and the **servo**. The **temperature** and **humidity** data are randomly generated by the Arduino's hardware. Listing 1 provides the first part of the source code for this Arduino program, in which we declare some useful symbolic constants and global variables.

Listing 1: Arduino Program (part 1)

```
1  #include <Servo.h>
2
3  const int BUFFER_SIZE = 100;
4  char buff[BUFFER_SIZE];
5
6  Servo servo;
7  const int servoPIN = 12;
8  const int servoMin = 100;
9  const int servoMax = 180;
10
11 const int publishDelay = 20000;   // publish every 20s
12 unsigned long myTime  = millis();
```

We're gonna need a buffer to store the incoming messages sent by the Gateway, so we reserve an array of characters whose length defined by the constant `BUFFER_SIZE` (line 3 and line 4). Line 6 through 9 of the listing are the global variables involved in the control of our `servo`. We will publish random temperature and humidity value every `publishDealy` milliseconds, and use the variable `myTime` to control the delay without actually freezing the big while loop (since the big while loop is also responsible for monitoring the incoming messages).

Listing 2: Arduino Program (part 2)

```
1  void setup() {
2    // setup built in led
3    pinMode(LED_BUILTIN, OUTPUT);
4
5    // setup servo
6    servo.attach(servoPIN);
7    servo.write(servoMin);
8
9    // set baud rate
10   Serial.begin(9600);
11
12   // seed the PRNG
13   randomSeed(analogRead(0));
14 }
```

Now we are ready to define the `setup()` function, as shown in Listing 2. The `setup()` function is straightforward. We use the symbolic constants defined earlier to setup the built-in LED and the pin that will control the servo. We also make the servo to rotate to the `servoMin` angle on launch (line 7). Our servo will only rotate between `servoMin` and `servoMax` degree. We then set the baud rate for the serial communication to the default value of Arduino's, which is 9600. The last step is to seed the built-in PRNG from which we will use to generate random integer value in later part of the implementation.

Listing 3 shows an (over) simplified implementation version of the `loop()` function.

Listing 3: Arduino Program (part 3)

```
1 void loop() {
2     if (Serial.available() > 0) {
3         // read from serial and process the received message
4     }
5     if (millis() - myTime > publishDelay) {
6         // send random temperature and humidity value to Serial
7     }
8 }
```

There are 2 events happens inside the forever loop. The first event occurs whenever the `Serial` instance detects incoming serial data coming from the Gateway. If this is the case, we will start process the character sequence to decode the message. The routine to accomplish this is explained in sub-section 2.1. The second event of the big loop occurs every a fixed amount of time, determined by the `publishDelay` variable. Currently, as shown in the listing, we are sending data packets to the Serial every 20 seconds. The `millis()` function is a built-in function that returns the number of milliseconds passed since the Arduino board began running the current program. To establish the 20s-delay inside the loop without hindering the serial's routine, we will utilise this handy function. We constantly ask for the elapsed time, and whenever the period surpassed the value stored in the `publishDelay` variable, we do the transmitting-data routine, which will be explained in subsection 2.2.

2.1 Decoding-Message Routine

Listing 4 shows what happens if a character sequence is detected at the serial.

Listing 4: Arduino Program (part 4)

```
1  if (Serial.available() > 0) {
2      // expected data pactket of the form: !<TOPIC_ENUM>:<VALUE>#
3      Serial.readBytesUntil('#', buff, BUFFER_SIZE);
4      String payload = String(buff);
5      if (payload.length() > 0) {
6        int code = payload.substring(payload.indexOf('!') + 1, payload.↩
             indexOf(':')).toInt();
7        int value = payload.substring(payload.indexOf(':') + 1, payload.↩
             indexOf('#')).toInt();
8        switch(code) {
9        case 0:
10         digitalWrite(LED_BUILTIN, (value == 1) ? HIGH : LOW);
11         break;
12       case 1:
13         servo.write(constrain(value, servoMin, servoMax));
14         break;
15       default:
16         break;
17       } // !switch
18     } // !payload > 0
19 } // !serial.available()
```

We use a method called `readBytesUntil()` to save the incoming character sequence in the `buff` array. We terminate the reading whenever we hit a #, as it marked the end of the message sequence. Having read the message as a C-style string, we construct a String instance for it and save the newly constructed object to the `payload` variable. The use of String here is to simplify further manipulation on the message sequence since the Arduino offer rich methods to handle String than that of raw C-style string.

It may seem redundant, but the first `if` coming right after the first string processing is to ensure we are not digging in an empty string, since we are about to invoke some expensive methods. We start off extracting the enumerated code that encodes the node that this message is targeting. It should be (for this moment) either the LED or the servo node. We will use 0 for LED and 1 otherwise. The next step is to extract the the data value that the Gateway wishes to enact on the target node and save it to the `value` variable. When done, we can happily use a simple `switch` statement on the `code` variable and process on according matches.

2.2 Transmitting-Data Routine

The following listing shows the activities happening every 20s-delay.

Listing 5: Arduino Program (part 5)

```
1 if (millis() - myTime > publishDelay) {
2     myTime = millis();  // reset current time
3     // using data frame of the form: !<ID>:<topic_name>:<data>:#
4     Serial.print("!1:temperature:" + String(random(0, 100)) + "#");
5     Serial.print("!1:humidity:" + String(random(0, 100)) + "#");
6 }
```

The code is straightforward. Whenever we hit the 20s interval, we reset the `myTime` variable to the current amount of elapsed time, and then start sending 2 random value (invoked through the `random()` function) for temperature and humidity.

Now that we've done with the first layer of our IoT system. The full implementation of this module can be found under the folder named *arduino-node* inside this Github repository.

3 Gateway Implementation

Things start to get more interesting and complicated here. The Gateway will have to implement both the MQTT protocol and UART communication, using Python. For the MQTT, in order to keep the project consistent, we will use the paho-mqtt package to establish the communication protocol. For the UART serial, we will stick with the popular pyserial package.

3.1 Gateway Overture

Before getting our hands dirty into the implementation of the Gateway, there are plentiful of things to discuss. We will devote this part of the Gateway section for sketching out the overall activities of our gateway program, as well as explaining some observations of the MQTT on Adafruit server.

3.1.1 Gateway Features

The bulk of this python program is to running a simple gateway that is capable of receiving data from the Adafruit server (broker) and dispatching it to an IoT Arduino node. The gateway also monitors the data sent from the Arduino and publishes it to the server, therefore establishing a form of communication between the broker and the sensor. The attributes of this simple implementation of our gateway are listed bellow.

- The gateway receives **temperature** and **humidity** data from the Arduino and publishes them to the broker.
- The gateway listens to **led** and **servo** data from the broker and dispatches them to the Arduino.
- The gateway is in charge of the **gtw-status** topic, which is exclusively reserved for the gateway to declare its status to the broker.
- The gateway also maintains a concrete connection with the Arduino. When launch, the gateway must ensure it has successfully communicated with the Arduino before establish the broker-connection. During runtime, if the arduino connection is compromised, the gateway will immediately terminate it activities, cut off its connection with the broker, do any necessary clean-up work, and exit with a warning.
- Whenever the gateway makes a successful connection to the broker, whether it is the first time the gateway connects to the server or it is a re-connection attempt due to any failure, it will automatically fetch the latest data from any topic it subscribes to and dispatch them to the Arduino.

With these features kept in mind, we can sketch out the behaviour of our gateway activities. The step-by-step implementation will be further demonstrated in later part of this section. In the mean time, it is not redundant to mention some "catches" when we use the Adafruit server in join with the paho-mqtt package.

3.1.2 Adafruit Limitation

MQTT is one of the powerful communication protocols, especially in the world of IoT. Since its invention in 1999, the latest MQTT version one can use at the point of writing this document is MQTT v5.0. While we can exploit many rich IoT features provided by the latest version of MQTT, that is not the case for the Adafruit broker, which by far supports just the MQTT v3.1.1. Even this version of MQTT is strong enough to handle almost any asspect of IoT communication, the Adafruit broker isolates itself from the standard and provides its own python API library for the end users. This API package is built as a wrapper of the paho-mqtt library, made possible by the people behind the Adafruit broker, to provide ease of use and discard the features of the standard that the Adafruit server is not capable of. So basically, if we use the paho-mqtt package to implement the MQTT and not the exclusive library offered by the Adafruit developers, some functionality provided by the library will not take effect. This is one significant point that need be kept in mind when doing paho-mqtt in joint with the Adafruit broker.

So what is missing from the Adafruit server? The comprehensive document (published by the Adafruit, of course) can be found here. They provide us with the guide to implement MQTT communication to the Adafruit server without using the exclusive package. There are many points to be considered, for example the limit of the rate of publishing messages, the must of unique client ID, the data packet format,... When doing MQTT with Adafruit server, it is best to follow the instructions in the published document to avoid undefined behaviour, or worse, account banning.

Why not using the exclusive library in the first place? As stated before, we need to keep the project consistent, because later on, we will have to use the paho-mqtt implementation anyway when we reach the last layer of our IoT system. What's more, the use of paho-mqtt is easy for future upgrade and scalability. We may want to switch to another modern broker other than Adafruit, and using the versatile package will not prevent us from doing so. In other words, we only explore a subset of what the paho-mqtt has in store for us, and we should aware of what the Adafruit broker is incapable of, for instance the *retained message* feature.

3.1.3 The Clean Session Problem

What is a clean session? When a client makes a connection to the broker, the client can make a decision whether its information should be remembered by the server. If the client chooses to have its information discarded by the broker (hence the connection session is clean), the server will not store any subscription information or undelivered messages for the client. This way of connection is sometimes called *non-persistent* connection, and is the connection type of choice for those clients that only publish data to the server. For those clients, on the other hand, that subscribe to some topics on the server, the MQTT provides a *persistent* connection, in which the broker will keep the information of the client (the client must provide a *client ID* so that the server can tell apart different clients).

The connection type, whether it is persistent or non-persistent, is up to the developer who

implements the MQTT client object. The Adafruit server supports this feature, with a warning up ahead:

> Connecting to Adafruit IO via MQTT and reusing the client ID of an existing connection will result in immediate disconnection of the other MQTT client. The MQTT specification requires that "each client connecting to the server has a unique ClientId." link

So if one opts to make her clients connection "unclean", then she has to look out for indistinguishable client IDs; otherwise, some of the clients will not make it to the Adafruit broker.

The clean session problem is mentioned here because it related to a more powerful feature of the MQTT protocol: the *retained message* feature. The retained message will only work for clients that set their clean session status to false. When a message published to a topic is retained, the broker will mark it as the latest data of that topic and will automatically publish that message to any persistent client reconnecting to the broker who previously subscribed to that same topic. Put it another way, if a client wants to fetch latest data (using the MQTT way) when it makes a successful connection to the server, the client must first need to be a **persistent client** and subscribe to topics of interest with **QoS = 1**. Any message that wants to get automatically published by the broker to the client statisfied these 2 requirement will also need be set as **retained** when published to according topic. The broker will only store the latest retained message for a topic and use it as the latest data. Retain makes writing basic MQTT-only Internet of Things clients easier, without it, a client that connects and subscribes to a feed topic has to wait until a new value is published on the feed to know what state it should be in.

This service provided by the MQTT protocol is just wonderful to use in place of the traditional HTTP routines, but unfortunately, the Adafruit server is not capable of implementing retained message, or, not implementing it the standard way:

> Among other factors, our scale, Adafruit IO's mix of MQTT HTTP APIs, the speed at which we're taking in new data, and the fact that we're already storing almost every message that is sent mean that a "simple" feature like retain becomes difficult to support without making MQTT service performance worse for everyone.

This means if we follow the same way use the retain service discussed above, in the case of Adafruit server, nothing will take effect. However...

3.1.4 The Retained Message Solution

The Adafruit does provide a solution for the retained message limitation, as this is one of the essential features of an IoT system. Using this solution, we will NOT gain the functionality of retain though, but we still get the latest message data without touching any HTTP protocol: the use of ***/get topic**:

> For any given Adafruit IO MQTT feed or group, subscribe to the appropriate topic using the feed or group key, then add /get to the topic you subscribed to and publish

anything to that new topic (our Arduino library uses the null character). IO will immediately publish, just for that client, the most recent value received on the feed.

And that's that. If we want any latest piece of data belong to any topic, we just simply subscribe to that topic and publish an empty message to the topic whose name is the said topic appended "/get". The client will immediately receive the latest data from the Adafruit server. However, there is a catch! The use of */get topic is not working for persistent connection. If a persistent client publishes to a /get topic, an undefined behaviour will occur. In other words, when working with Adafruit server, we should ignore the existence of the clean session service and leave it set as default (non-persistent) to avoid weird outcome.

Okay, with these knowledge at hands, we are ready to step into the gateway program implementation. As usual, the full code is sitting under the folder named *gateway* in this Github repository.

3.2 Step-by-Step Implementation

The code is presented as listings, and only those that need explained will be included.

3.2.1 Setup Part

It is always a best practice to define and use symbolic constants instead of "magic" data. Listing 6 shows the initial part of our gateway program.

Listing 6: Gateway Program - Initial Setup

```
import paho.mqtt.client as mqtt
import time, queue, serial, threading
from serial.serialutil import SerialException
import serial.tools.list_ports as serialtool

# ------- CONFIGURATIONS & CONSTANTS SETUP ------------
# Broker Configurations
HOST_NAME = "io.adafruit.com"    # using Adafruit IO server
HOST_PORT = 8883                 # secure connection
USERNAME  = "***"        # replace with actual broker username
PASSWORD  = "***"        # replace with actual authentication password

# Gateway Configurations
GATEWAY_ID  = "GTW002"
GROUP_KEY   = "iot-lab"
PUB_QOS = 1
SUB_QOS = 1
STAT_TOPIC  = "gateway-status"
SUBSCRIBE_TOPICS = [
    "arduino-led",       # record led status
```

```
21      "arduino-servo",    # record change in servo angle
22      STAT_TOPIC          # listen to disconnect signal from the broker
23  ]
24  SCAN_DELAY  = 1.0
25  TIME_OUT    = 5.0       # maximum waiting time for blocking
26  MAX_FAILED_ATTEMPTS = 3
27
28  # Devices Configurations
29  BAUDRATE    = 9600    # Arduino default baudrate
30  COM_TOKEN   = 'USB Serial Device'
31
32
33  # ---------------------- COLOR CODES --------------------
34  class col:
35      cdtag   = '\033[35;1m'
36      mestag  = '\033[1;34m'
37      pubtag  = '\033[31;1m'
38      subtag  = '\033[0;33m'
39      good    = '\033[92m'
40      bad     = '\033[31m'
41      user    = '\033[0;95m'
42      topic   = '\033[1;37m'
43      message = '\033[0;94m'
44      stage   = '\033[0;93m'
45      esc     = '\033[0m'
```

The majority of the symbolic constants in the listing are self-explained by their names. Here we have a `col` class which is used to highlight the debug messages, as shown in Figure 2. Each debug message has a preceding tag key to make it stand out. We are having a problem here. The broker refuses to grant our client access to the server, as told by the red lines. Notice that the lines in red were not hard-coded. Each unsuccessful attempt to the broker has a reason message associated with it, and the paho-mqtt API provides us with a utility method `connack_string()` to get the descriptive cause of the failure. The use of this method is saved until we reach the `callbacks` implementation.

It is worth mentioning that we are making 3 attempts to connect to the server before exiting the program. The constant `MAX_FAILED_ATTEMPTS` at line 26 determines this number of retrying attempts. Another interesting point is that the topics involved in our IoT system belong to a *group* of topics called **iot-lab** (line 15). The use of group will be further explained in section 4. Right now we should focus on the list of subscribe topics at line 19. Because we are listening to the LED and servo topics (as described in subsection 3.1), we should include their topic name into this list. Notice that the **gtw-status** topic is included using another symbolic constant. This will come in handy later, since we treat this topic different from the others. This will also be explained until we reach the later part of our gateway implementation.

```
PS X:\Projects\2021\adafruit-simple-iot\gateway> py gateway.py
Finding serial port...
Attached to: COM4
[C] GTW002 failed to connect to ***: Connection Refused: not authorised.
[D] GTW002 disconnected from ***
[WARNING] Unexpected Disconnection --> The connection was refused.
[C] GTW002 failed to connect to ***: Connection Refused: not authorised.
[D] GTW002 disconnected from ***
[WARNING] Unexpected Disconnection --> The connection was refused.
[C] GTW002 failed to connect to ***: Connection Refused: not authorised.
[D] GTW002 disconnected from ***
[WARNING] Unexpected Disconnection --> The connection was refused.
Maximum connection attempts tried. Quiting...
PS X:\Projects\2021\adafruit-simple-iot\gateway>
```

Figure 2: Connection Failed: Not Authorised

```
PS X:\Projects\2021\adafruit-simple-iot\gateway> py gateway.py
Finding serial port...
[WARNING] no COM port found
Quitting...
PS X:\Projects\2021\adafruit-simple-iot\gateway>
```

Figure 3: No COM port found

Let us run the program again, but this time without the connection of the Arduino as COM port 4. The result is depicted in Figure 3. Our gateway complains that it found no attached device at the terminal end, so it quits immediately. Later on we will see that, any unexpected disconnection between the gateway and the Arduino will cause our program to terminate without hesitation.

3.2.2 Global Variables and Shared Variables

Listing 7: Gateway Program - Variables

```
1  # ---------- GLOBAL VARIABLES ---------------
2  block       = False # flag to block thread
3  exquit      = False # flag to raise disconnect interrupt
4  greeting    = True  # indicate if this is the first connection
5  reinit      = False # flag to call init() procedure on reconnect
6  texceed     = False # indicate if we've reached maximum blocking time
7  failcount   = 0     # count attempts on fail activity
8  msg         = ''    # hold buffer message read from serial
9
10 # ----------- SHARED-MEMORY -----------------
11 messageBuffer = queue.Queue()   # holds incoming messages
12 devicesBuffer = queue.Queue()   # holds outgoing messages
13 terminate     = False   # control thread termination
14 scanning      = False   # flag to indicate initial setup
15 timerexit     = False   # control timer termination
```

The next part of our program is reserved for declaring the global and shared variables, as Listing 7 shows. We have quite a lot variables here, each has it own responsibility on certain aspect of the program, though most of them are involved in the controlling of asynchronous threads. Why thread? The paho-mqtt API offers multiple callbacks that get automatically invoked after a certain action of the MQTT protocol. These threads are, however, asynchronous to the main thread of our program, so we will need these variables to prevent messy debug messages on the console.

Aside from the variables, we have 2 instances of type `queue` acting as buffers. These queues will ease up the rate of transmitting data and handle multiple incoming messages without hindering other activities in the big while loop. It is important to note that the rate of transmission is NOT unlimited when it comes to Adafruit server:

> Adafruit IO's MQTT server imposes a rate limit to prevent excessive load on the service. If a user performs too many publish actions in a short period of time then some of the requests will be rejected and an error message will be published on your /throttle topic. The current rate limit is at most 30 requests per minute for free accounts, 60 per minute with an IO+ account, and expandable via Adafruit IO+ Boost applied to your account.

You heard the IO, we must have some control in the case the Arduino sends too many data packets in a mimute. The `deviceBuffer` queue will hold these messages from the Arduino. The Arduino can flood this queue with as many data packets as it wants, because our Gateway program will process these data from the queue one by one independent of the Artduino rate, thus controlling the rate of publishing to the server. On the other hand, the `messageBuffer` will take on the job of holding messages published by the broker. Its purpose is the same as of the `deviceBuffer`: prevent the contention occurs inside the big while loop.

3.2.3 Defining Callbacks

We will need to define some callbacks that are part of the MQTT API protocol. Some of them serve as debugging, while the others are essential to the gateway functionality. Listing 8 shows the definition of the `on_connect()` callback.

Listing 8: Gateway Program - on_connect()

```
1 def on_connect(client, userdata, flags, rc):
2     if rc != 0: # connection failed
3         print(f"{col.cdtag}[C]{col.esc} {GATEWAY_ID} failed to connect↩
              to {USERNAME}: {col.bad}{mqtt.connack_string(rc)}{col.esc}↩
             ")
4     else:       # connect successfully
5         global greeting
6         if greeting:    # the first time this client get connected
```

```
7             print(f"{col.cdtag}[C]{col.esc} {GATEWAY_ID} connected to ↩
                  {col.user}{USERNAME}{col.esc} with result: {col.good}{↩
                  mqtt.connack_string(rc)}{col.esc}")
8             greeting = False
9             global block
10            block = False
11        else:           # client is making re-connection attempt
12            print(f"{col.cdtag}[C]{col.esc} Reconnected to {col.user}{↩
                  USERNAME}{col.esc} with result: {col.good}{mqtt.↩
                  connack_string(rc)}{col.esc}")
```

The passed in argument `rc` is used to check the status of connection. If the client failed to connect to the server (`rc != 0`), then we print out the reason of failure using the method `connack_string()` to interpret the reason code. If everything is fine, we continue to see whether the client is making the first connection or a re-connection attempt, and process accordingly.

Listing 9: Gateway Program - on_disconnect()

```
1 def on_disconnect(client, userdata, rc):
2     print(f"{col.cdtag}[D]{col.esc} {GATEWAY_ID} disconnected from {↩
          col.user}{USERNAME}{col.esc}")
3     if rc != 0:
4         print(f"{col.bad}[WARNING]{col.esc} Unexpected Disconnection ↩
              --> {mqtt.error_string(rc)}")
5         global failcount, reinit, scanning
6         reinit = True       # the program should do the reinit when ↩
              get reconnect
7         scanning = False    # terminate all activites in the big while↩
              loop
8         failcount = failcount + 1   # count retrying attempts
```

In contrast, Listing 9 shows the routine that get called when our client disconnects from the broker. If the disconnection were unexpected (network drops, etc), then we would expect the program to reconnect to the server after a while. The `reinit` and `scanning` flag are set to reflect this expectation. We also count the consecutive fail attempts inside this callback for making termination decision on other part of the program.

The next callback shall be the `on_message()`, represented in Listing 10. This time we utilize the argument `userdata` which is automatically passed in by the MQTT implementation. The `userdata` is an object reserved for whatever purpose the developers want to pass custom data into callbacks that accept the argument. Here, we will use it to filter out the owning messages (messages that are sent and received by the same client). Before publishing any message to the server, we set the `userdata` object to hold an array of string where the first string is the payload and the second string is the topic name. This instance of the `userdata` object will then get

passed into any callback that has a parameter accepting the object, including our `on_message()` callback. It is then the job of the `on_message()` function to use the passed in argument to check if the received message is the same as one hold by the `userdata`.

Listing 10: Gateway Program - on_message()

```
def on_message(client, userdata, message):
    msgv = message.payload.decode()                          # get message value
    msgt = message.topic[message.topic.find('.') + 1:]       # get message topic
    # filter out owning message
    if msgv == userdata[0] and msgt == userdata[1]:
        global block    # release block
        block = False
    else:
        print(f"{col.mestag}[M]{col.esc} Received {col.message}{msgv}{col.esc} on topic: {col.topic}{msgt}{col.esc}")
        # if receive offline instruct from the broker
        if (msgt == STAT_TOPIC and msgv == "offline"):
            global exquit
            exquit = True

        # otherwise add message to the buffer
        else:
            if scanning: messageBuffer.join()
            messageBuffer.put((msgt, msgv))
```

The listing reveals more than just the filtering-message thing. Here we also handle the case where we receive an **offline** message from the topic **gtw-status**. The flag `exquit` is set to indicate we should clean up and terminate our program.

The 2 last callbacks involving in this program are shown in Listing 11. For the publishing, we just simply check the topic we are sending message to and change the debug message accordingly. The subscribing, on the other hand, has a job to release the thread lock hold by the variable `block`. Why blocking thread? This has to do with the order that the debug messages appear on the console. Normally, the subscribing protocol is very fast and less error-prone, so the subscription message will usually get printed first, even before the successful connection message get printed. Blocking is put in here to make the debug console more intuitive.

Listing 11: Gateway Program - on_publish() and on_subscribe()

```
def on_publish(client, userdata, mid):
    if "get" in userdata[1]:
        print(f"Fetching {col.stage}latest data{col.esc} from {
```

```
PS X:\Projects\2021\adafruit-simple-iot\gateway> py gateway.py
Finding serial port...
Attached to: COM4
[C] GTW002 connected to _angelus with result: Connection Accepted.
[S] Subscribed to: arduino-led
Fetching latest data from arduino-led...
[S] Subscribed to: arduino-servo
Fetching latest data from arduino-servo...
[S] Subscribed to: gateway-status
GTW002 has set its status to online
Monitoring...
```

Figure 4: A successful gateway connection.

```
            userdata[1][:-4]}...")
    elif userdata[1] is STAT_TOPIC:
        print(f"{GATEWAY_ID} has set its status to {col.stage}{userdata[0]}{col.esc}")
    else:
        print(f"{col.pubtag}[P]{col.esc} Published {col.pubtag}{userdata[0]}{col.esc} to: {col.topic}{userdata[1]}{col.esc}")

def on_subscribe(client, userdata, mid, granted_qos):
    global block
    print(f"{col.subtag}[S]{col.esc} Subscribed to: {col.topic}{userdata}{col.esc}")
    block = False    # release block
```

Before moving on, let see how a successful connection to the server looks like. Figure 4 illustrates a normal gateway connection. We start off finding the COM port where the Arduino attachs to, and then try to establish a connection to the server. On successful connection, we start fetching latest data from the topics in the subscribe list. Finally, we publish an **online** message to the **gtw-status** topic before jump into the forever while loop.

3.2.4 Utility Functions

Let's talk a little bit about some helpful functions used throughout the program. There are quite a lot of them, but the document only lists those that worth mentioning. Listing 12 shows the `init()` routine, which is responsible for subscribing to the required topics and getting latest data upon each successful subscription. This function will also come in handy when our client get unexpected disconnection and then wish to resubscribe to the topics after reconnecting to the server. The function also illustrates the use of `userdata` object, the */get solution for latest data fetching, and the variable `block` to synchronize debug message. The function finishes off setting the gateway status to **online** by publishing the according message to the `STAT_TOPIC`.

Listing 12: Gateway Program - init() function

```
def init():
    global gtw, ser
    # subscribe to required topics
    # -----
    for top in SUBSCRIBE_TOPICS:
        global block
        block = True    # raise a block flag
        gtw.user_data_set(top)
        gtw.subscribe(topic=f"{USERNAME}/feeds/{GROUP_KEY}.{top}", qos=SUB_QOS)
        while block:
            pass        # block thread until fully subscribed
        # fetch latest data, except for the 'status' topic
        if top is not STAT_TOPIC:
            publish('', f"{top}/get")

    # publish 'online' to status feed
    # ------
    publish("online", STAT_TOPIC)
```

Listing 13 shows the implementation of the wrapper `publish()` function. We need a wrapper here because there are a couple things that need be done besides publishing. The method `wait_for_publish()` that comes right after the execution of the `publish()` method has the effect of synchronization. It will block the current thread until the callback `on_publish()` get invoked. We also have to manually blocking the current thread if we sending message to the topic we subscribe to. The client must detect the owing message and filter it out before it can continue its execution. This blocking might take forever, although it may only happen in rare case. Even so, the function is provided with mechanism to automatically release the block if the waiting time exceeds the value of the constant `TIME_OUT`. We achieve this by launching a separate thread that acts like a countdown timer. The shared variable `texceed` is the flag that will be set to indicate if the waiting time is too long. The timer routine that run on the separate is shown in Listing 14.

Listing 13: Gateway Program - publish() function

```
def publish(payload, topic):
    global gtw
    gtw.user_data_set([payload, topic])
    pub = gtw.publish(f"{USERNAME}/feeds/{GROUP_KEY}.{topic}", payload, PUB_QOS)
    pub.wait_for_publish()
    # if publishing to subscribed topic
    if topic in SUBSCRIBE_TOPICS:
```

```
8           global block, texceed, timerexit
9           # then block thread until having fully filtered out the owning↩
                message
10          block = True
11          texceed = False
12          timerexit = False
13          taskTiming = threading.Thread(target=timer) # only block for ↩
               the maximum time decided by the timer
14          taskTiming.start()
15          while block:
16              if texceed: # timer exceeds
17                  print(f"{col.bad}[WARNING]{col.esc} Reached maximum ↩
                        waiting time for validating own message. Continuing↩
                        ...")
18                  taskTiming.join()
19                  block = False
20          if taskTiming.is_alive():   # join the separate timer thread
21              timerexit = True
22              taskTiming.join()
```

One of the most buggy thing about multi-threaded programming is that we usually forget to join the separate thread. Our `publish()` function does its best to join any thread its launch on every possible exit path.

Listing 14: Gateway Program - timer() function

```
1  def timer():
2      global texceed
3      count = 0
4      while True:
5          if timerexit:
6              break
7          time.sleep(1.0)
8          count = count + 1
9          if count == TIME_OUT:
10             texceed = True
11             break
```

The implementation of `timer()` is pretty straightforward. We use the built-in `sleep()` method to count the elapsed second. There are 2 possible ways our timer can terminate: whther the gateway filters out its owning message on time or it does not. Either way, we make sure the while loop is broken so that the thread can exit properly. The shared data `timerexit` is used here to signal the if the gateway has filtered out the owing message on time.

Listing 15: Gateway Program - watcher() function

```
def watcher():
    global exquit, ser
    while True:
        if terminate:
            ser.close()
            break
        try:
            driver()
        except SerialException:
            print(f"{col.bad}[WARNING]{col.esc} Lost connection to ↩
                serial. Quitting...")
            ser.close()
            exquit = True
            break
```

Finally, there is another co-routine that monitors the incoming data from the Arduino, the `watcher()` function (Listing 15). Much like the timer, we have a shared variable `terminate` to signal if the our thread should exit on command. As long as this variable is not asserted, the watcher will call the `driver()` function repeatedly to check for character sequences at the terminal. There is another exit point for the watcher: the Arduino get disconnected unexpectedly. When this happens, the `driver()` will raise an exception, and the watcher signals our main thread to terminate the gateway program.

3.2.5 Main Script

It's time to put everything we've come through together. Listing 16 shows the main script of our gateway program.

Listing 16: Gateway Program

```
# ------------------ MAIN GATEWAY SCRIPT -------------------

# instantiate a gateway client and setup some options
# -----
gtw = mqtt.Client(client_id=GATEWAY_ID)
gtw.tls_set_context()
gtw.username_pw_set(username=USERNAME, password=PASSWORD)
gtw.will_set(f"{USERNAME}/feeds/{GROUP_KEY}.{STAT_TOPIC}", 'offline')

# register callbacks
# -----
gtw.on_connect = on_connect
gtw.on_disconnect = on_disconnect
gtw.on_publish = on_publish
```

```
gtw.on_subscribe = on_subscribe
gtw.on_message = on_message

# connect to serial com port
# -----
try:
    print(f"{col.stage}Finding serial port...{col.esc}")
    ser = serial.Serial(getport(), BAUDRATE)
except SerialException:
    print("Quitting...")
    exit(1)

# start external thread
# -----
taskWatching = threading.Thread(target=watcher)
taskWatching.start()

# connect to the broker
# -----
gtw.connect(host=HOST_NAME, port=HOST_PORT)

# start recording change
# -----
gtw.loop_start()

# block thread until having connected successfully
# -----
block = True
failcount = 0
while block:
    if failcount == MAX_FAILED_ATTEMPTS:
        terminate = True
        taskWatching.join()
        gtw.disconnect()
        time.sleep(0.1)
        print("Maximum connection attempts tried. Quiting...")
        exit(1)

# subscribe to required topics & publish 'online' to status feed
# -----
init()

# monitor external devices
# -----
print(f"{col.stage}Monitoring...{col.esc}")
```

```
60 scanning = True
61 try:
62     while True:
63         # do the init procuder again on reconnect
64         if gtw.is_connected and reinit:
65             init()
66             reinit = False
67             scanning = True
68 
69         # if there still be data waiting to be published in the buffer
70         while gtw.is_connected and not reinit and not devicesBuffer.↩
               empty():
71             tup = devicesBuffer.get()
72             publish(tup[1], tup[0])
73             devicesBuffer.task_done()
74 
75         # if there still be message waiting to be processed in the ↩
               buffer
76         while not messageBuffer.empty():
77             tup = messageBuffer.get()
78             ser.write(f"!{SUBSCRIBE_TOPICS.index(tup[0])}:{tup[1]}#".↩
                   encode())
79             messageBuffer.task_done()
80 
81         # constantly check if there is a terminate signal from broker
82         if exquit:
83             raise KeyboardInterrupt
84 
85 
86         # scan delay
87         time.sleep(SCAN_DELAY)
88 
89 # disconnect on cancel
90 # -----
91 except KeyboardInterrupt:
92     # terminate watcher thread
93     print("Terminating watcher...")
94     terminate = True
95     taskWatching.join(timeout=1.0)
96     publish("offline", STAT_TOPIC)
97     res = gtw.disconnect()
98     time.sleep(0.1) # delay a small amount of time for on_disconnect()↩
           callback
99     print(f"Disconnect Status: {col.good if res == 0 else col.bad}{↩
           mqtt.error_string(res)}{col.esc}")
```

I believe with the comment in between, the listing is self-conveying. There is one point that need mentioned, however. Because our gateway will run forever on end inside the big while loop (starting at line 62), if we want to safely terminate the program (via Ctrl + C, for example), we should wrap the big loop in side a `try` block and have an exception handler to terminate our program clean and neat (line 91).

3.3 Gateway Conclusion

So that's how we implement the gateway. I especially focus on making the program more reliable, with various unexpected behaviour handled properly. Our program, unfortunately, has a small bug. This bug is not attributed to the developer (of course), but rather it is because we lack an useful feature of the MQTT on Adafruit: the *will set*. If a client registers a backup message (a will) on a topic, then when this client unexpectedly disconnects from the server, the broker will automatically publish this registered message to that topic. An example use of will set can be found at line 8 of the main program above. Here we tell the broker that if the gateway unexpectedly looses connection to the server, then the broker should publish the message **offline** to the **gtw-status** topic, to indicate that the gateway has (badly) gone offline. But we are unlucky because the Adafruit server does not provide such a handy feature, and we should start to smell trouble. If our gateway unexpectedly gets disconnected from the server, then we have no way to publish the offline status without the help of will set. This means at some point in the lifetime of our gateway, the status topic shows that our gateway is online, but in fact it may not be working due to unexpected connection error. Our gateway might find its own way to reconnect to the server after a fatal disconnection, but still this is unacceptable for system that demands higher level of reliability. If this is the case, then we should consider state-of-the-are brokers like the Eclipse Mosquitto or the HiveMQ. These come with the MQTT v5.0 support, which is much more suitable for critical IoT projects. The Adafruit, however, has its own advantages. It provides beautiful interface monitors called *dashboards* to help us organize the components involved in the system (section 4). Also, the security on the Adafruit is extremely reliable. If the user's private key get accidentally leaked (public source code on Github, for example), then the user automatically receives an email from the Adafruit team with a friendly warning. All and all, Adafruit should be a place of choice for education and small-scaled projects.

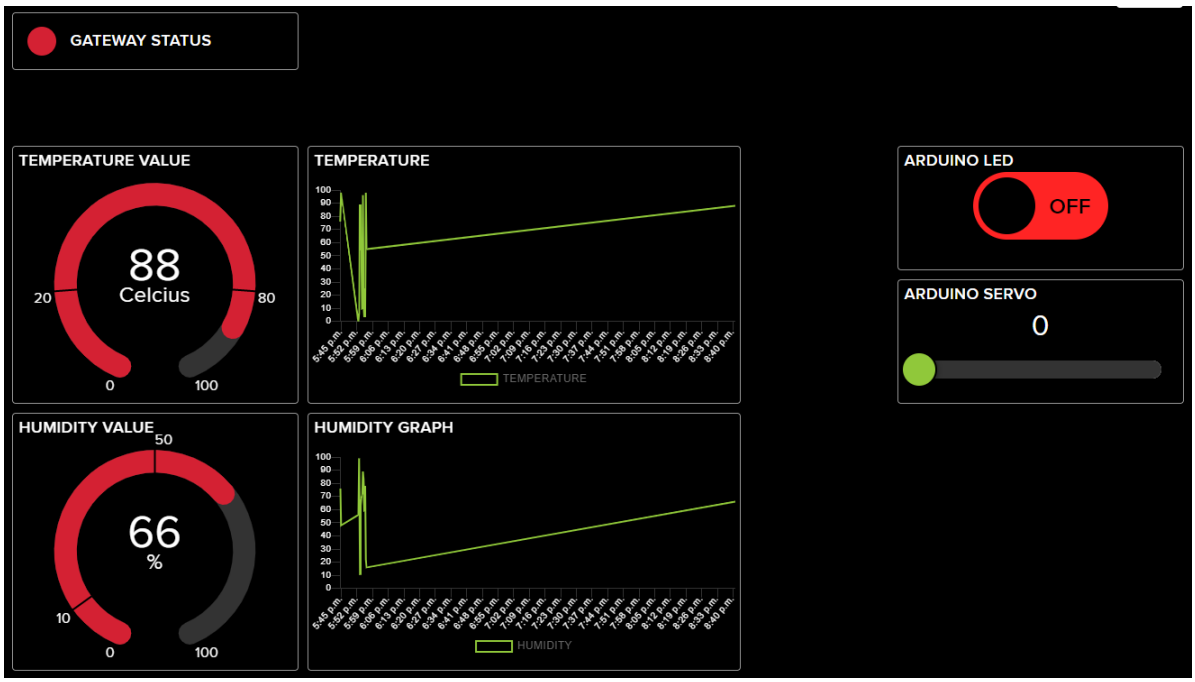

Figure 5: Dashboard

4 Server Configuration

This should be a short (break) section, in which I showcase the dashboard for the project on the Adafruit server and discuss a bit about grouping on Adafruit. Figure 5 illustrates the dashboard design for this project. Dashboard design is easy and straightforward, so no further explanation is required.

Figure 6 captures the grouping feature of the Adafruit IO. Currently I have 2 groups, 1 is created default by the server, and 1 for this project. Grouping makes related topics more organized and eases the management of users. When a topic is assigned to a group other than the Default group, its proper name is prefix with its group name and a colon.

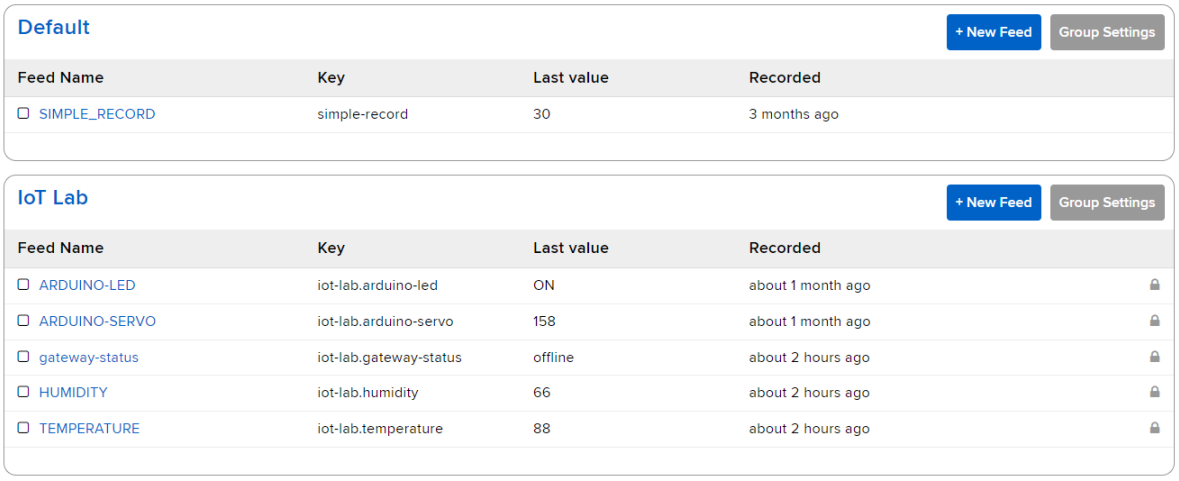

Figure 6: Adafruit Grouping

5 Monitoring Application

We hit another mother lode. This time we have to bring all the MQTT client implementation onto a mobile application. Most of the idea will be the same as what have been proposed for the gateway. I will spend more time discuss about modern mobile application development rather than digging into the MQTT. Let warm up with the Jetpack components.

5.1 Jetpack Architecture

Until recently, Google did not recommend a specific approach to building Android apps other than to provide tools and development kits while letting developers decide what worked best for a particular project or individual programming style. That changed in 2017 with the introduction of the Android Architecture Components which, in turn, became part of Android Jetpack when it was released in 2018. Android Jetpack consists of Android Studio, the Android Architecture Components and Android Support Library together with a set of guidelines that recommend how an Android App should be structured. The Android Architecture Components are designed to make it quicker and easier both to perform common tasks when developing Android apps while also conforming to the key principle of the architectural guidelines.

There are many components that makes up the Jetpack architecture, and our mobile application only use three: the *ViewModel*, the *Data Binding*, and the *Navigation Graph*. They will be discussed shortly in upcoming sub-sections. Before moving on, it is important to understand the Jetpack approach to app development is not mandatory. While highlighting some of the shortcoming of other techniques that have gained popularity of the years, Google stopped short of completely condemning those approaches to app development. Google appears to be taking the position that while there is no right or wrong way to develop an app, there is a recommended way.

5.1.1 Traditional Architecture

An Android project typically consists of a single activity which contains all of the code for presenting and managing the user interface together with the back-end logic of the app. Up until the introduction of Jetpack, the most common architecture followed this paradigm with apps consisting of multiple activities (one for each screen within the app) with each activity class to some degree mixing user interface and back-end code.

This approach led to a range of problems related to the lifecycle of an app (for example an activity is destroyed and recreated each time the user rotates the device leading to the loss of any app data that had not been saved to some form of persistent storage) as well as issues such as inefficient navigation involving launching a new activity for each app screen accessed by the user.

At the most basic level, Google now advocates single activity apps where different screens are loaded as content within the same activity. Modern architecture guidelines also recommend separating different areas of responsibility within an app into entirely separate modules (a concept

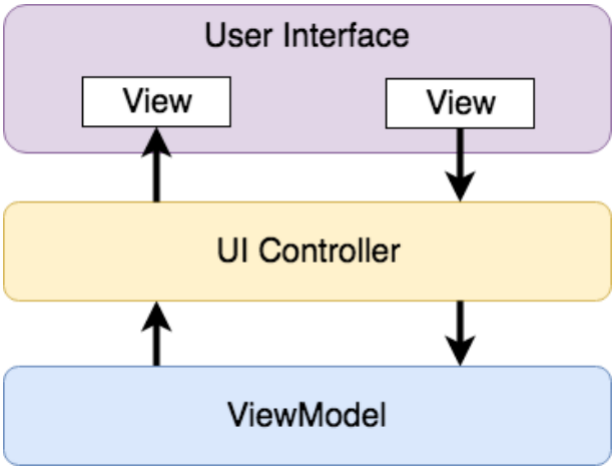

Figure 7: Jetpack Component - ViewModel

Google refers to as "separation of concerns"). One of the keys to this approach is the *ViewModel* component.

5.1.2 The ViewModel Component

The purpose of ViewModel is to separate the user interface-related data model and logic of an app from the code responsible for actually displaying and managing the user interface and interacting with the operating system (Figure 7). When designed in this way, an app will consist of one or more **UI Controllers**, such as an activity, together with ViewModel instances responsible for handling the data needed by those controllers.

In effect, the ViewModel only knows about the data model and corresponding logic. It knows nothing about the user interface and makes no attempt to directly access or respond to events relating to views within the user interface. When a UI controller needs data to display, it simply asks the ViewModel to provide it. Similarly, when the user enters data into a view within the user interface, the UI controller passes it to the ViewModel for handling.

This separation of responsibility addresses the issues relating to the lifecycle of UI controllers. Regardless of how many times a UI controller is recreated during the lifecycle of an app, the ViewModel instances remain in memory thereby maintaining data consistency. A ViewModel used by an activity, for example, will remain in memory until the activity completely finishes which, in the single activity app, is not until the app exits.

5.1.3 LiveData and Data Binding

Consider an app that displays realtime data such as the current price of a financial stock. There are only two ways that the UI controller can ensure that the latest data is displayed in the user interface. One option is for the controller to continuously check with the ViewModel to find out if the data has changed since it was last displayed. The problem with this approach, however, is that it is inefficient. To maintain the realtime nature of the data feed, the UI controller would

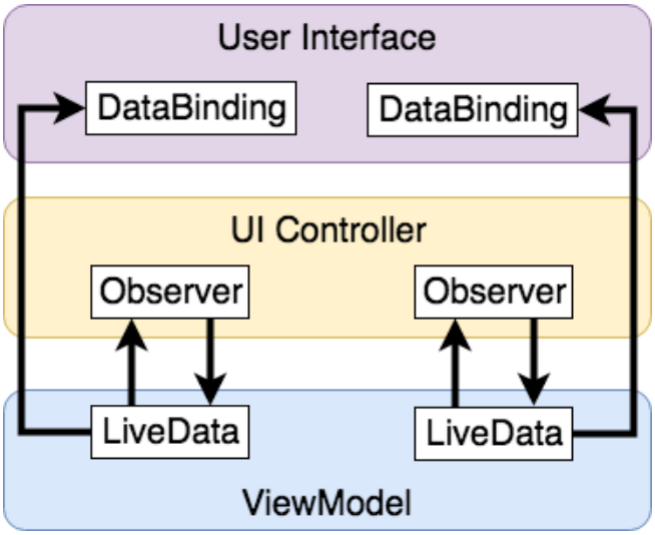

Figure 8: Jetpack Component - Data Binding

have to run on a loop, continuously checking for the data to change.

A better solution would be for the UI controller to receive a notification when a specific data item within a ViewModel changes. This is made possible by using the *LiveData* component. LiveData is a data holder that allows a value to become **observable** . In basic terms, an observable object has the ability to notify other objects when changes to its data occur thereby solving the problem of making sure that the user interface always matches the data within the ViewModel.

Another of the key advantages of using LiveData is that it is aware of the lifecycle state of its observers. If, for example, an activity contains a LiveData observer, the corresponding LiveData object will know when the activity's lifecycle state changes and respond accordingly. If the activity is paused (perhaps the app is put into the background), the LiveData object will stop sending events to the observer. If the activity has just started or resumes after being paused, the LiveData object will send a LiveData event to the observer so that the activity has the most up to date value. Similarly, the LiveData instance will know when the activity is destroyed and remove the observer to free up resources.

Android Jetpack includes the Data Binding Library which allows data in a ViewModel to be mapped directly to specific views within the XML user interface layout file. Data binding allows the LiveData value stored in the ViewModel to be referenced directly within the XML layout file avoiding the need to write code to keep the layout views updated. Figure 8 illustrates the idea.

5.1.4 Navigation Graph

Prior to the introduction of Android Jetpack, the implementation of navigation within an app was largely a manual coding process with no easy way to view and organize potentially complex navigation paths. This situation has improved considerably, however, with the introduction of the Android Navigation Architecture Component combined with support for navigation graphs in Android Studio.

A navigation graph is an XML file which contains the destinations that will be included in

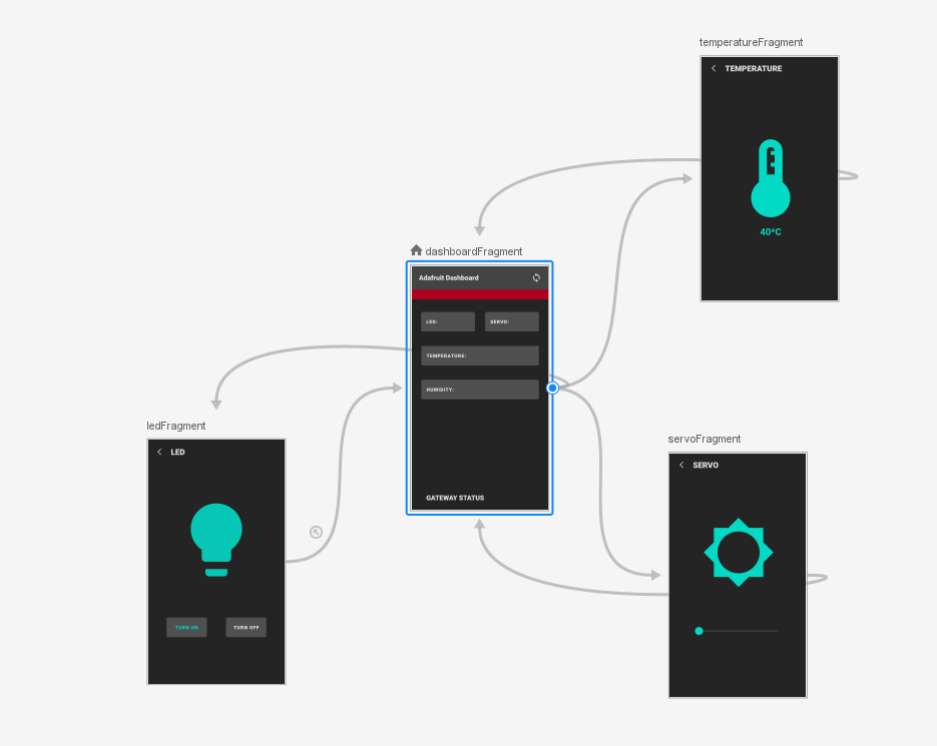

Figure 9: Jetpack Component - Navigation Graph

the app navigation. In addition to these destinations, the file also contains navigation actions that define navigation between destinations, and optional arguments for passing data from one destination to another. Android Studio includes a navigation graph editor that can be used to design graphs and implement actions either visually or by manually editing the XML. Figure 9 shows the navigation graph of our mobile application.

5.2 MQTT in Mobile Application

It would a pain if the implementation of our Android application is presented here. As I stated before, the project is hosted in this Github repository, so it will be much more practical (and realistic) to pay it a visit. I will devote this section to explain how I achieve the MQTT protocol in an mobile app, together with modern components that we have come through. Listing 17 shows the SIMPLIFIED version of the main activity class of our mobile app.

Listing 17: Main Activity

```
1 public class MainActivity extends AppCompatActivity
2 {
3     // private fields
4     // -----
5     private MqttAndroidClient _client;
6     private MqttConnectOptions _connectOptions;
7     private DisconnectedBufferOptions _disconnectedBufferOptions;
8
```

```
9       private DashboardViewModel _dashboardViewModel;
10
11
12      // lifecycle methods
13      // -----
14      @Override
15      protected void onCreate(@Nullable Bundle savedInstanceState) {
16          super.onCreate(savedInstanceState);
17
18          // inflate root view for this activity using view bing library
19          setContentView(ActivityMainBinding.inflate(getLayoutInflater()↩
                ).getRoot());
20
21          // setup view model for fragments
22          _dashboardViewModel = new ViewModelProvider(this).get(↩
                DashboardViewModel.class);
23
24          // initialize and connect a client
25          initClient();
26          connectClient();
27      }
28
29      @Override
30      protected void onDestroy() {
31          super.onDestroy();
32          disconnectClient();
33      }
34
35      // mqtt utilities
36      // -----
37      private void initClient() {
38          final String serverURI = BrokerConfig.AUTH_CONTEXT + "://" + ↩
                BrokerConfig.HOST_NAME + ':' + BrokerConfig.HOST_PORT;
39          _client = new MqttAndroidClient(getApplicationContext(), ↩
                serverURI, ClientConfig.CLIENT_ID);
40          setupClient();
41          setupCallback();
42      }
43
44      private void setupCallback() {
45          _client.setCallback(new MqttCallbackExtended() {
46              @Override
47              public void connectComplete(boolean reconnect, String ↩
                    serverURI) {
48                  Timber.tag(logTag).d("Connected successfully: %s", ↩
```

```
                serverURI);
49          }
50
51          @Override
52          public void connectionLost(Throwable cause) {
53              if (cause != null) {
54                  Timber.tag(logTag).e("Lost connection to broker: %s", cause.getMessage());
55                  _dashboardViewModel.setErrorMessage(cause.getMessage() + _dashboardViewModel.errorSuffix);
56                  _dashboardViewModel.setConnected(false);
57              }
58          }
59
60          @Override
61          public void messageArrived(String topic, MqttMessage message) {
62              Timber.tag(logTag).d("Received '%s' from %s", message, topic.substring(topic.indexOf('.') + 1));
63              if (topic.contains(ClientConfig.SUBSCRIBE_TOPICS[ClientConfig.TOPICS.LED.ordinal()])) {
                    // led message
64                  _dashboardViewModel.setLedStatus(message.toString());
65              }
66              else if (topic.contains(ClientConfig.SUBSCRIBE_TOPICS[ClientConfig.TOPICS.SERVO.ordinal()])) {
                    // servo message
67                  _dashboardViewModel.setServoValue(message.toString());
68              }
69              else if (topic.contains(ClientConfig.SUBSCRIBE_TOPICS[ClientConfig.TOPICS.TEMPERATURE.ordinal()])) {
                    // temperature message
70                  _dashboardViewModel.setTemperatureValue(message.toString());
71              }
72              else if (topic.contains(ClientConfig.SUBSCRIBE_TOPICS[ClientConfig.TOPICS.HUMIDITY.ordinal()])) {
                    // humidity message
73                  _dashboardViewModel.setHumidityValue(message.toString());
74              }
75              else if (topic.contains(ClientConfig.SUBSCRIBE_TOPICS[
```

```
                    ClientConfig.TOPICS.STATUS.ordinal()])) {     // ↩
                    status message
76                      _dashboardViewModel.setGtwStatus(message.toString↩
                            ());
77                  }
78              }
79
80              @Override
81              public void deliveryComplete(IMqttDeliveryToken token) {
82                  try {
83                      if (!token.getMessage().toString().isEmpty()) {
84                          Timber.tag("mqtt").d("Delivered successfully (↩
                                messageID = %d)", token.getMessageId());
85                      }
86                  } catch (MqttException e) {
87                      e.printStackTrace();
88                  }
89              }
90          });
91
92      }
93
94      private void setupClient() {
95          // setup connection options
96          _connectOptions = new MqttConnectOptions();
97          _connectOptions.setUserName(BrokerConfig.USERNAME);
98          _connectOptions.setPassword(BrokerConfig.PASSWORD.toCharArray↩
                ());
99          _connectOptions.setAutomaticReconnect(false);
100         _connectOptions.setCleanSession(true);
101
102         // setup disconnected buffer options
103         _disconnectedBufferOptions = new DisconnectedBufferOptions();
104         _disconnectedBufferOptions.setBufferEnabled(true);
105         _disconnectedBufferOptions.setBufferSize(100);
106         _disconnectedBufferOptions.setPersistBuffer(false);
107         _disconnectedBufferOptions.setDeleteOldestMessages(false);
108     }
109
110     private void connectClient() {
111         _dashboardViewModel.setConnecting(true);
112         try {
113             _client.connect(_connectOptions, null, new ↩
                    IMqttActionListener() {
114                 @Override
```

```
115
116                 public void onSuccess(IMqttToken asyncActionToken) {
117                     _client.setBufferOpts(_disconnectedBufferOptions);
118                     subscribe();
119                     fetchLatest();
120                     _dashboardViewModel.setConnected(true);
121                     _dashboardViewModel.setConnecting(false);
122                 }
123
124                 @Override
125                 public void onFailure(IMqttToken asyncActionToken, ↩
                        Throwable exception) {
126                     Timber.tag(logTag).e("%s failed to reach %s: %s", ↩
                            ClientConfig.CLIENT_ID, BrokerConfig.HOST_NAME,↩
                             exception.getMessage());
127                     _dashboardViewModel.setErrorMessage(exception.↩
                            getMessage() + _dashboardViewModel.errorSuffix)↩
                            ;
128                     _dashboardViewModel.setConnected(false);
129                     _dashboardViewModel.setConnecting(false);
130                 }
131             });
132         } catch (MqttException e) {
133             e.printStackTrace();
134         }
135     }
136
137     private void disconnectClient() {
138         if (_client.isConnected()) {
139             try {
140                 IMqttToken token = _client.disconnect();
141                 token.setActionCallback(new IMqttActionListener() {
142                     @Override
143                     public void onSuccess(IMqttToken asyncActionToken)↩
                            {
144                         Timber.tag(logTag).d("%s disconnected ↩
                                successfully", ClientConfig.CLIENT_ID);
145                     }
146
147                     @Override
148                     public void onFailure(IMqttToken asyncActionToken,↩
                            Throwable exception) {
149                         Timber.tag(logTag).e("%s failed to disconnect:↩
                                 %s", ClientConfig.CLIENT_ID, exception.↩
                                getMessage());
```

```
150                     }
151                 });
152 
153             } catch (MqttException e) {
154                 e.printStackTrace();
155             }
156         }
157     }
158 
159     private void subscribe() {
160 
161         for (String topic : ClientConfig.SUBSCRIBE_TOPICS) {
162             try {
163                 String realTopic = String.format("%s/feeds/%s.%s", ↩
                        BrokerConfig.USERNAME, ClientConfig.GROUP_KEY, ↩
                        topic);
164                 _client.subscribe(realTopic, 0, null, new ↩
                        IMqttActionListener() {
165                     @Override
166                     public void onSuccess(IMqttToken asyncActionToken)↩
                            {
167 
168                     }
169 
170                     @Override
171                     public void onFailure(IMqttToken asyncActionToken,↩
                            Throwable exception) {
172                         Timber.tag(logTag).e("Failed to subscribe to %↩
                            s", topic);
173                     }
174                 });
175             } catch (MqttException e) {
176                 e.printStackTrace();
177             }
178         }
179     }
180 
181     private void fetchLatest() {
182         for (String topic : ClientConfig.SUBSCRIBE_TOPICS) {
183             publish(String.format("%s/get", topic), "");
184         }
185     }
186     private void fetchLatest(String topic) {
187         publish(String.format("%s/get", topic), "");
188     }
```

```
189
190     private int publish(String topic, String msg) {
191         String realTopic = String.format("%s/feeds/%s.%s", ↩
                BrokerConfig.USERNAME, ClientConfig.GROUP_KEY, topic);
192         int msgID = 0;
193         try {
194             IMqttToken token = _client.publish(realTopic, msg.getBytes↩
                    (StandardCharsets.UTF_8), 0, false);
195             token.setActionCallback(new IMqttActionListener() {
196                 @Override
197                 public void onSuccess(IMqttToken asyncActionToken) {
198
199                 }
200
201                 @Override
202                 public void onFailure(IMqttToken asyncActionToken, ↩
                        Throwable exception) {
203                     Timber.tag("mqtt").e("Failed to publish %s to %s",↩
                            msg ,topic);
204                 }
205             });
206             msgID = token.getMessageId();
207         } catch (MqttException e) {
208             e.printStackTrace();
209         }
210         return msgID;
211     }
212 }
```

All the MQTT-things are gathered inside this class. The implementation may look different from that of in Python, but the overall idea is the same. When the host activity is created, we make the 2 calls to the `initClient()` and `connectClient()` function. They respectively construct an MQTT client instance and establish a connection for it to the broker (line 25 and line 26). When the activity get destroyed, we also disconnect the client from the server, as shown at line 32. Another point that need pointed out is that we override the callback `onSuccess()` at line 115 to have it subscribe to required topics and fetch latest data for us each time we successfully connect to the server.

And I think that's the main idea. Again, all the Android-related things can be found at this Github repository. Overall it's kinda fun to work with mobile, it really shapes up the organization mindset for the developers.

6 Conclusion

It's time to wrap things up. I must say this seemingly small-scaled project gave me much more of what I expected. It may hard to catch up at first but when the pace is maintained, one could push it far beyond expectation. After all, I would like to give my professor, Mr. Nhan, a big thankful appreciation, for his wonderful dedication to the teaching of this course.

References

[Web] Arduino, https://www.arduino.cc/

[Web] MQTT Retained Message, http://bitly.ws/mvqS

[Web] MQTT QoS, http://bitly.ws/mvr4

[Book] Neil Symth, Android Studio 4.2 Development Essentials - Java Edition