Drew Seidel
Roy Kravitz
ECE 558
Saturday, March 4, 2023

Project #2 Design Report

Objective:

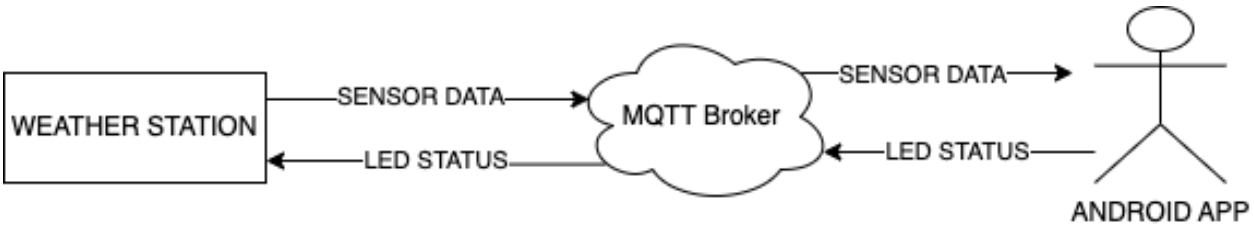
The objectives of this weather app were the following:

- Gain more experience with Android development
- Become familiar with MQTT and IoT development
- Develop a weather station that:
    - collects and publishes to an MQTT topic, temperature, gas resistance, humidity, air pressure, and altitude
    - subscribes to and MQTT topic, and update LED(s) accordingly
- Develop an Android App that:
    - subscribes to the MQTT topic(s) that have published the sensor readings from the weather station
    - Publishes the LED(s) status that the weather station receives
- Gain experience utilizing fragments for a multi-screen user interface experience

Overall Design Hierarchy:

Using HiveMQ as the MQTT broker, both the Android App, and the Pico W Weather Station can publish and subscribe to topics.

So, what is required is the following:



To accomplish this, two topics can be utilized, formatted as JSON objects:

1. The weather station publishes to the topic ‘drew/weather_station and the Android App subscribes to this topic. The topic will contain data about air pressure (hPa), altitude (M), humidity (%), gas (Ω), and temperature (˚F). The current implementation leaves the units as is, but of course, post processing could be done on either the app side or firmware side in future implementations. For example, air quality can be computed using the currently available metrics, and that could be displayed instead
2. The Android app publishes to the topic ‘drew/led_status_update’ and the weather station subscribes to this topic. This topic will contain information for the weather station about whether the board LED should be on or off, and whether the externally hooked up LED should be on or off. The current implementation does the following:

a. User controls the board LED using a switch in the Android App
b. External LED turns on if the temperature is greater than 70˚F

On the HiveMQ Web Client Broker, when subscribed to both topics, the way I formatted the JSON object literals is apparent.

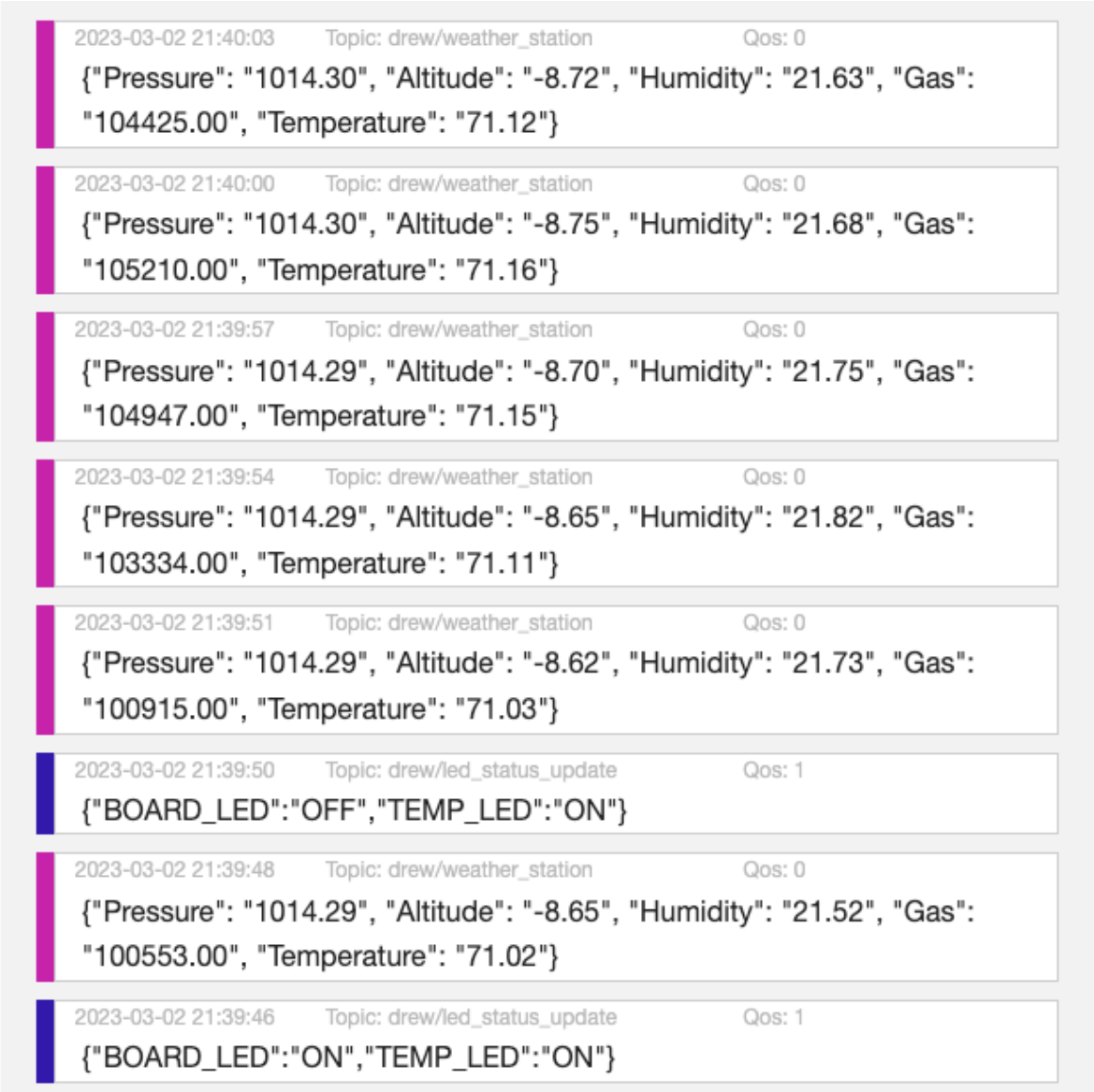


Figure 1: Subscribing to topics on HiveMQ MQTT Broker

For postprocessing capability in future editions, the data strings from the sensor have been sent solely as numbers and have left the app responsible for adding units and so forth. This makes it easy to convert to integers or doubles if desired and leaves it easy to display the values as strings.

Weather Station Design:

The weather station utilizes:

- A Raspberry Pi Pico W
- A red external LED (with current limited resistor)
- A BME680 module that collects all readings
- An I^2C connection between the Pico W and the BME680
- Firmware developed in Python using MicroPython

The following figures reveal a few key snippets from the firmware:

The infinite forever loop simply reads the sensors, creates the string that will be used to publish to the topic for the and Android app, and checks for new messages from the subscribed to topic that contains information about the LEDs.

```
try:
    client = mqtt_connect()
except OSError as e:
    reconnect()
# main loop
while True:
    #  display measurements on console. Offset by -5 degrees for PCB temp
    print("\nTemperature: %0.1f F" % ((bme680.temperature - 5) * 9/5 + 32) )
    print("Gas: %d ohm" % bme680.gas)
    print("Humidity: %0.1f %%" % bme680.humidity)
    print("Pressure: %0.3f hPa" % bme680.pressure)
    print("Altitude:  %0.2f meters" % bme680.altitude)
    topic_msg = weather_update()                    #call weather update to format
string
    client.publish(topic_pub, json.dumps(topic_msg)) #publish topic message as JSON
    client.check_msg()                              #check subscription message
    time.sleep(0.2)                                 #200ms
```

Figure 2: Main python loop

Formatting the Python dictionary that will be converted to string upon publication with `json.dumps()` and the JSON library for Python.

```
def weather_update():
    topic_msg = {
              f"Temperature":  f"{((bme680.temperature - 5) * 9/5 + 32):.2f}",
              f"Gas":  f"{bme680.gas:.2f}",
              f"Humidity":  f"{bme680.humidity:.2f}",
              f"Pressure": f"{bme680.pressure:.2f}",
              f"Altitude": f"{bme680.altitude:.2f}"
              }
```

```
    return topic_msg
```

Figure 3: Formatting string for JSON dump in publication

The LED handler is callback handler for when new messages have been published by the Android App containing information about the LED status. The handler converts the JSON object into a Python dictionary and can then be easily utilized to turn off or on the corresponding LED. Note that messages are only published to this topic by the Android App if there is a relevant change, that way, the weather station isn't needlessly spending its time on functions where there are is no status update to process.

```
#handle LEDs callback to subscription led_topic_sub
def handle_leds(topic, msg):
    if (topic == led_topic_sub):  #should always be true, but check in case
        message = json.loads(msg) #convert JSON object to Python Dictionary
        if (message["BOARD_LED"] == "ON"):
            board_led.on()
            print("User turning on board LED")
        else:
            board_led.off()
            print("User turning off board LED")
        if (message["TEMP_LED"] == "ON"):
            temp_led.on()
            print("Temperature above 70˚F. Turning on temp LED")
        else:
            temp_led.off()
            print("Temperature dropped below 70˚F. Turning off temp LED")
```

Figure 4: Callback handler for LED status topic subscription

Android App Design:

The Android App using fragments to include a connect screen, and main weather screen.

The main activity manages the two fragments and allows them to replace each other upon certain actions (the connect button tapped on the first fragment, and the disconnect button on the second fragment).

The UI of the App is the following:

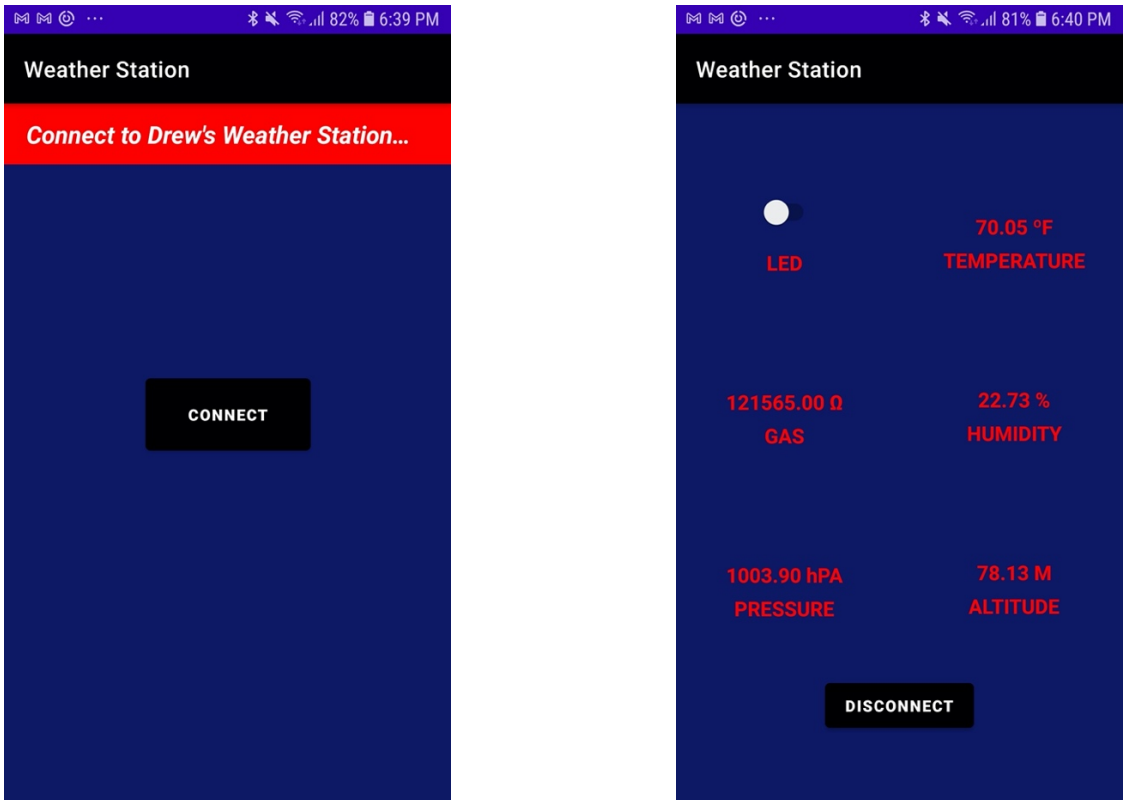


Figure 5: User interface for the Android App

The main activity manages the two full-screen fragments by using a defined communicator interface with two functions:

```
interface Communicator {
    fun viewData(connected: Boolean)
    fun disconnect()
}
```

Figure 6: Interface for Communicator. Override methods are created in MainActivity.

With the simplicity of the first fragment (connect screen), the MainActivity only needs to help manage transaction replacements of the two fragments.

```
//upon connect button clicked on main screen, go to second fragment for
weather station
//interaction
override fun viewData(connected: Boolean)
{
    if (connected){
        val transaction = this.supportFragmentManager.beginTransaction()
        val viewDataFragment = ViewDataFragment()
        transaction.replace(R.id.view_manager, viewDataFragment)
        transaction.addToBackStack(null)
        transaction.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE)
        transaction.commit()
    }
}

//upon disconnect button pressed in second fragment, go back to the home
screen
override fun disconnect() {
    val transaction = this.supportFragmentManager.beginTransaction()
    val connectFragment = ConnectFragment()
    transaction.replace(R.id.view_manager, connectFragment)
    transaction.addToBackStack(null)
    transaction.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE)
    transaction.commit()
}
```

Figure 7: Implementation of override functions responsible for changing the fragment displayed.

```
// go to view data method in Main Activity to change fragments
private fun connectButtonClicked() {
    communicator.viewData(true)
}
```

Figure 8: viewData method of communicator called to replace connect screen with data screen upon the connect button pressed.

```
private fun disconnectButtonClicked() {
    mqttClient.disconnect() //disconnect from MQTT
    communicator.disconnect() //use the communicator to call the disconnect
method in Main Activity
}
```

Figure 9: Disconnection from MQTT and disconnect method upon disconnect pressed

In the ViewDataFragment, upon successful connection to MQTT, upon MqttCallback, the topic ID is checked, and all the corresponding data is updated on screen. Furthermore, we check if the temperature as risen below above 70˚F and updated the Boolean representing whether the temperature is too high if necessary.

```
//MqttCallback object provides the main brains for the frontend UI
//upon MQTT message arrival, we parse the JSON formatted string,
//update the display, check the temperature, and if there was a
//change from the previous state, update highTemp, and call the ledHandler
object : MqttCallback {
```

```
    @SuppressLint("SetTextI18n")
    override fun messageArrived(topic: String?, message: MqttMessage?) {
        val msg = "Received message: ${message.toString()} from topic:
$topic"
        Log.d(TAG, msg)

        // since a message arrived I'm assuming that the topic string is not
null
        // update all weather data from JSON formatted incoming string
        if (topic!! == WEATHER_UPDATE) {
            val weatherData = JSONObject(message.toString())
            binding.updateTemp.text = weatherData.getString("Temperature") +
" °F"
            binding.updateGas.text = weatherData.getString("Gas") + " Ω"
            binding.humidityUpdate.text = weatherData.getString("Humidity")
+ " %"
            binding.pressureUpdate.text = weatherData.getString("Pressure")
+ " hPA"
            binding.altitudeUpdate.text = weatherData.getString("Altitude")
+ " M"

           val tempString = weatherData.getString("Temperature")
           val temp : Double = tempString.toDouble()

            if ((temp >= 70) && !highTemp) {
                highTemp = true
                ledHandler(LED_STATUS_UPDATE)
            } else if ((temp < 70) && highTemp) {
                highTemp = false
                ledHandler(LED_STATUS_UPDATE)
            }
        }
    }
```

Figure 10: Main UI update brains of the App.

Note that we simply get strings from JSON formatted strings and can then easily parse through and directly update the app text accordingly. Then we check the weather and call the handler for the LEDs if necessary (if the state has switched from below 70 to equal to or above 70, or vice versa).

That brings us to the implementation of handling and publishing the LEDs status. Three variables, global and private to the fragment classed are utilized to keep track of the LED state.

```
private var highTemp : Boolean = false  //true if over 70 f, false otherwise
private var boardLedOn : Boolean = false //true if led switch turned on,
false otherwise, initialized to false
private lateinit var led_STATUS : String //led string to be JSON formatted
to send to Raspberry Pico W, containing status of LEDS
```

Figure 11: LED status variables

It was shown how a change in highTemp, will call ledHandler(*LED_STATUS_UPDATE*), but what about boardLedOn that is controlled by the switch on the App?

```
//create method for handling the switch for the onboard Pico W LED
binding.LEDSwitch.setOnCheckedChangeListener { _, isChecked ->
    boardLedOn = isChecked
    ledHandler(LED_STATUS_UPDATE)
}
```

Figure 12: Handler for the App switch

For this, binding.LEDSwitch.setOnCheckedChangeListener is utilized, and isChecked is true when the switch is on, and false when the switch is off.

Upon a change, the ledHandler() is called, the topic message is updated, and published.

```
//handle LED and send publish topic as JSON formatted string
private fun ledHandler(topic: String){

    val boardLEDStatusString = if (boardLedOn) "ON" else "OFF"
    val tempLEDStatusString = if (highTemp) "ON" else "OFF"

    if (mqttClient.isConnected()) {
        val messageJSON = JSONObject()
        messageJSON.put("BOARD_LED", boardLEDStatusString)
        messageJSON.put("TEMP_LED", tempLEDStatusString)
        val message = messageJSON.toString()
        mqttClient.publish(
            topic,
            message,
            1,
            false,
            object : IMqttActionListener {
                override fun onSuccess(asyncActionToken: IMqttToken?) {
                    val msg = "Successfully published topic: $topic"
                    Log.d(TAG, msg)
                }

                override fun onFailure(
                    asyncActionToken: IMqttToken?,
                    exception: Throwable?
                ) {
                    val msg =
                        "Failed to publish: to topic: $topic exception:
${exception.toString()}"
                    Log.d(TAG, msg)
                }
            })
    } else {
        Log.d(TAG, "Impossible to publish, no server connected")
    }
}
```

Figure 13: Updating LED status topic, formatted as JSON, and sending as a string

Since there is only one topic to publish to, the message is updated here. In future editions, if more topics are wished to be published to, both the topic and message could be passed into a publication method such as this one (but then the message would need to already be processed elsewhere).

Challenges and Improvements:

The project provided a good exercise in creating a full-fledged IoT system, gaining experience in MQTT, and making design decisions. Some of the key challenges faced were:

- Fragment usage and management. Even in a simple use case as this one, they turned out to be a challenging hurdle.
- Layout management. With all the readings I collected, I resorted to text display with a friendly looking UI, as using more widgets would have eliminated the ability to display a sensor reading. In future editions, more fragments could be added to increase the usage of cool widgets.
- JSON usage to package data. While in the end this made the code a lot neater, and the data packets much more organized, it was a hurdle to get used to at first but make updating a lot easier.
- Errors caused due to dependency issues, or missing service in the AndroidManifest.xml for example. These errors caused a spam on console errors that were difficult to parse through.

Overall, I thought the project provided a great starting point for creating applications for IoT embedded designs. One thing that could make the introduction fragments easier, is releasing multiple apps using the varying ways utilizing the various ways that they can be used (full screen transitions, smaller fragments, navigation, etc.) in addition to the released fragment example.

References:

1. BME 680 Driver: https://github.com/robert-hh/BME680-Micropython
2. ../ece558w23_proj2_release/*
3. *../FragmentExample/* (ECE 558)*
4. https://www.tutorialspoint.com/send-data-from-one-fragment-to-another-using-kotlin
5. *http://www.hivemq.com/demos/websocket-client/*
6. *https://www.tomshardware.com/how-to/send-and-receive-data-raspberry-pi-pico-w-mqtt*