In God we trust

School of Computer Engineering

Sharif University of Technology

Embedded Systems

Project Report

Smart Garage Remote

Professor Mohsen Ansari

Presented by: Saee Saadat, Neda Taghizadeh,

Negar Nobakhti, Hamila Mailee

Contents

Table of Figures

Introduction

In this project, we design a framework that enables us to control our garage door with a remote application using the internet. To this end, we set up a server, and from the user's side, a message containing desired instructions will be sent to this server. The protocol for transferring these messages is MQTT, and NodeMCU is the intermediary that receives the client's messages. A pair of transmitter/receiver modules are needed to interpret the received signals and open/close the door. The state machine below shows an overall view of our project.

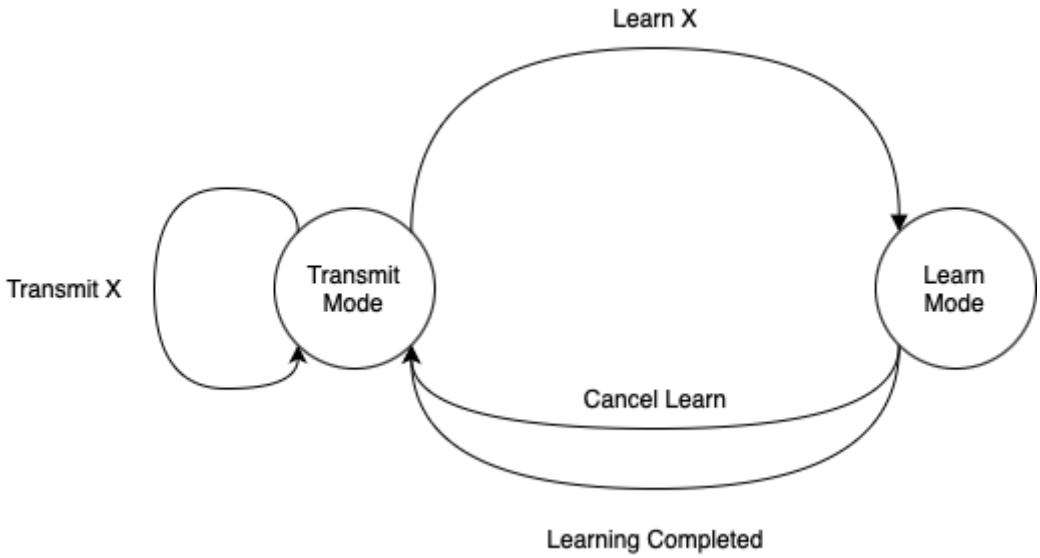

Figure 1 – The state machine representing our project.

Internet Connections

As mentioned in the introduction, the protocol used for connection between server and clients is MQTT. In this section, we first provide a brief description of this protocol. After that, we introduce our server and client and how they connect.

MQTT

MQTT is a TCP-based protocol and the standard in IoT messaging due to its reliability and transfer speed. After setting up the MQTT broker, the clients who are in charge of sending data (in our case, users pressing buttons) will define some "Topics", and the other clients (RF Transmitter) will subscribe to the ones related to them. One advantage of MQTT is that it allows the client to not only receive a message but also send it. In our project, the messages transmitted between different parts are in JSON form, following the below format: (the array form in the JSON representations below indicates the possible options for messages rather than the exact definition of arrays)

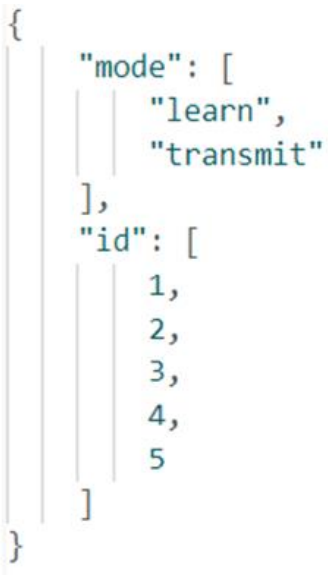

Figure 2 - Messages from client to server (to reach the end devices).

The mode “learn” is for the initiation of the connection, saving the values for opening/closing the door in the memory of our NodeMCU. In the “transmit” mode, we will use the saved values in memory to operate accordingly.

Figure 3 - Messages from devices to server (reporting the state of devices after receiving a signal).

The mode “error” means that the id had no value, or the learning process was not successful. “message” mode is equivalent to success and is mostly used for tracking the actions performed by end devices. The “message” field is an extra explanation that distinguishes events, e.g., in the “message” mode, this field can be “Signal for id: 3 has been learned”, or “Signal transmitted successfully”.

Server

For our MQTT broker, we use EMQX.io which provides services in two types of cloud and enterprise. The cloud version is a fully managed MQTT service with a free trial for 14 days, which is suitable for our case.

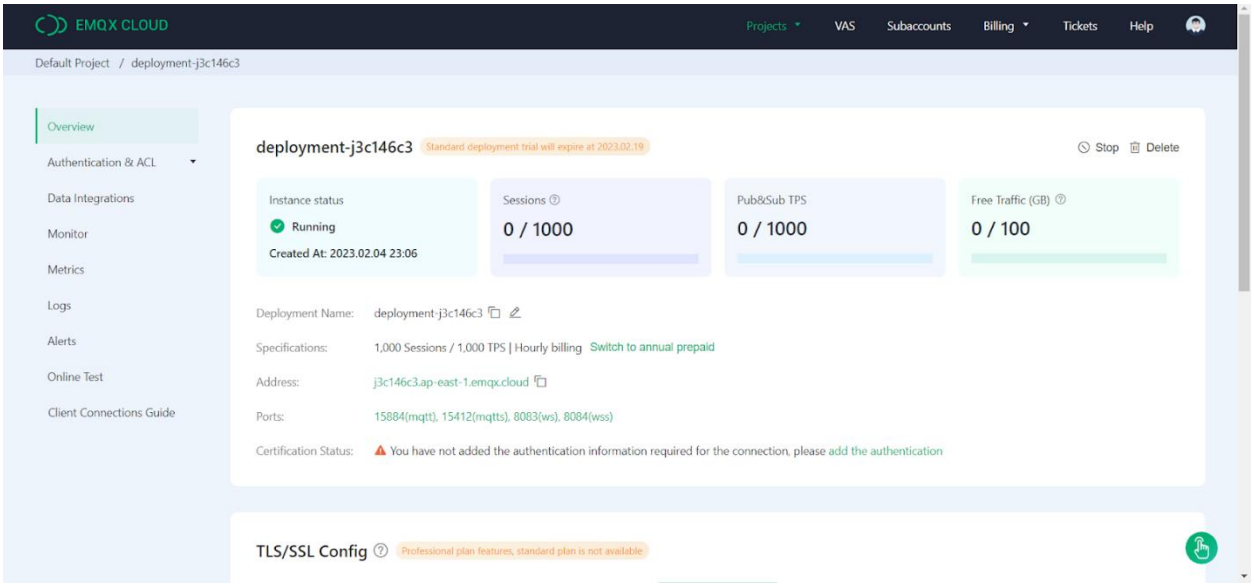

Figure 4 - An overview of the MQTT server provided by EMQX Cloud

From the Authentication section, users can be added by assigning usernames and passwords for each.

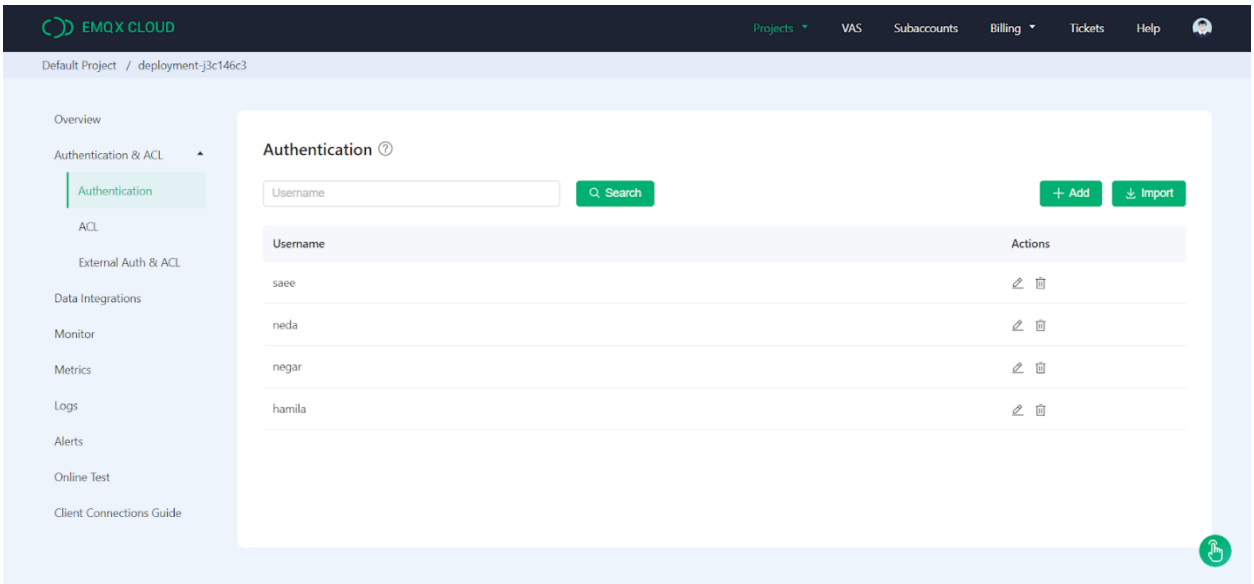

Figure 5 - Authenticated users that can subscribe/publish

Client

Our users can connect to the whole kit using an IOS mobile application. After connecting to Wi-Fi and NodeMCU, users will be able to manage 5 different channels, each responsible for a radio signal controlling a door. The figures below show the implementation of our application.

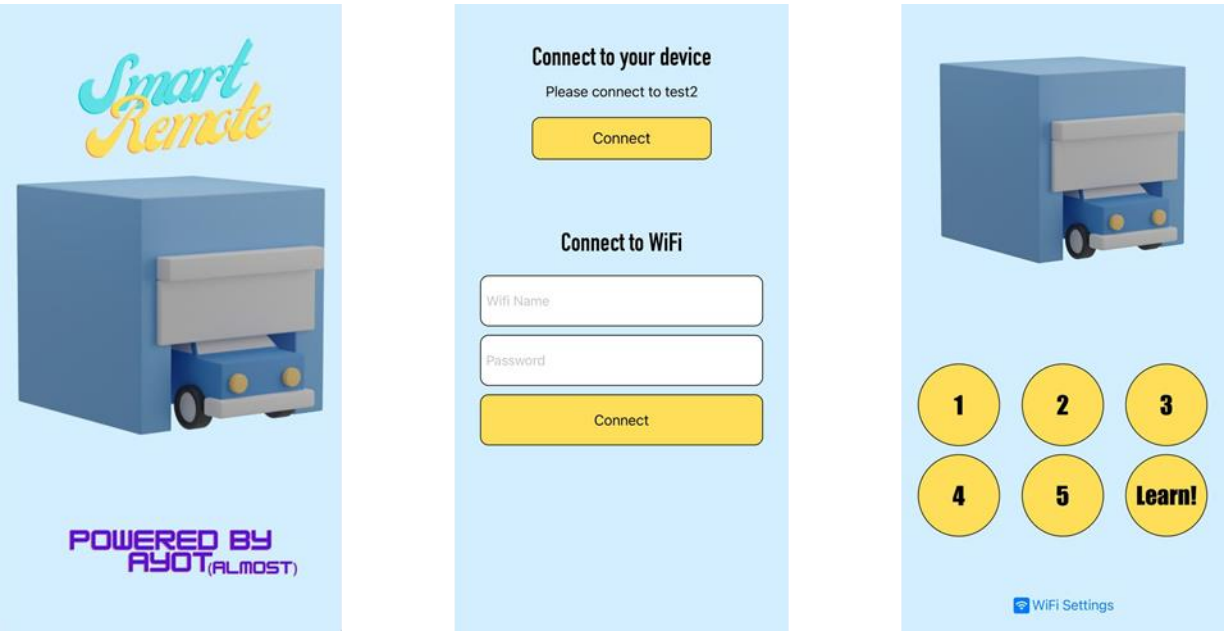

Figure 6 - Start page, login/connect page, and the control page from left to right.

After connecting to NodeMCU and establishing the connection, some of the sent/received messages tracked by the MQTT broker are shown below:

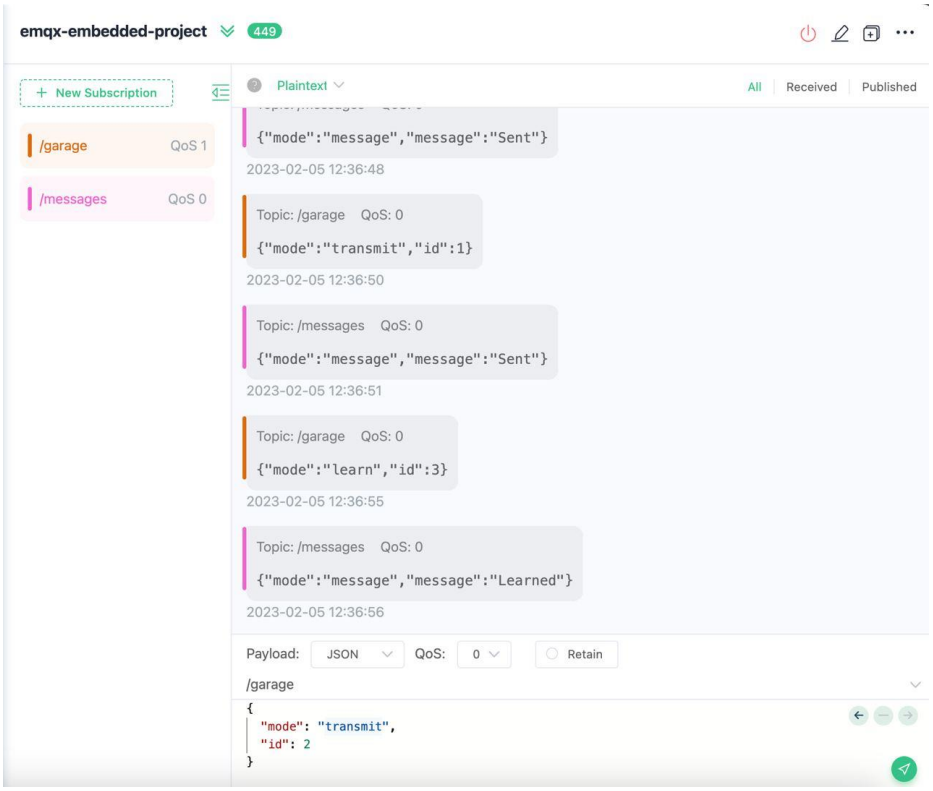

Figure 7 - Messages transferred between garage and application.

The topic “garage” is used for the instructions that are sent from the application, and the “messages” topic shows the signals sent from RF modules to our broker.

Stabilizing Connection

Our device will save the initial SSID and password, and upon turning it on, it will try to connect based on these previously saved values. While the device is trying to connect, it will act as an access point itself, enabling us to change the connection (connecting to a new Wi-Fi and establishing the connection from scratch). The reset button added to our circuit works using the function below:

```
void checkForResetButton() {
  if (digitalRead(resetButtonPin) == HIGH) {
    // reset network info
    Serial.println("Resetting");
    preferences.remove("ssid");
    preferences.remove("pass");
    WiFi.disconnect();
    setupAP();
  }
}
```

Figure 8 – Reset button’s function.

When the reset button is pushed, the saved values will be cleared, and the device awaits information about a new connection (SSID and password) from a new client (IOS client).

Hardware Modules

The needed modules for the hardware section are NodeMCU, RF Transmitter, and RF Receiver. A description of each one is provided in this section.

NodeMCU

NodeMCU is an open-source software and hardware development environment, containing the crucial elements of a computer: CPU, RAM, and networking (Wi-Fi). These properties make it an excellent choice for Internet of Things (IoT) projects of all kinds.

To check the functionality of our chip, we programmed a simple code to turn an LED on for 1 second and then turn it off in a loop. The code for this program and a picture of its implementation are shown below.

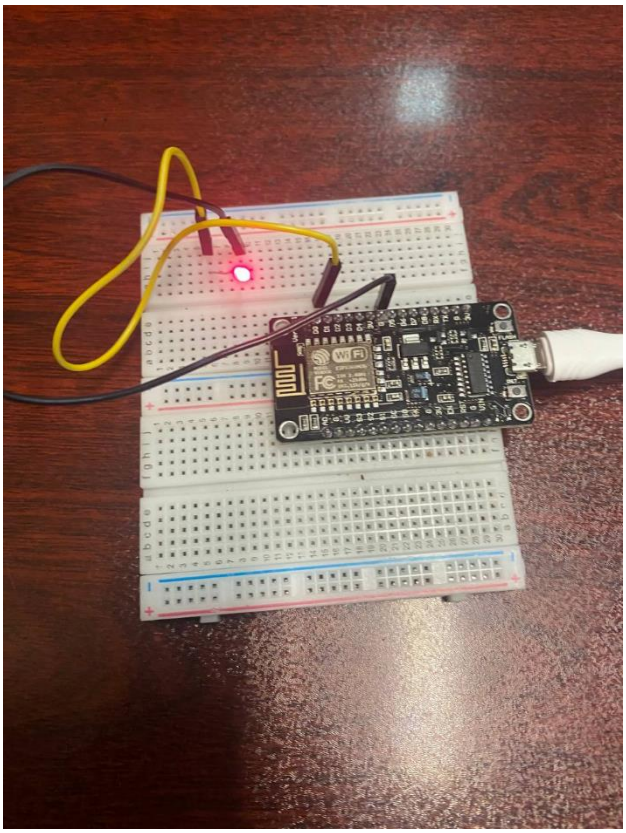

Figure 9 – Using NodeMCU to light the LED periodically.

```
int ledPin = D0;

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(ledPin, OUTPUT);
  Serial.print("SETUP\n");
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(ledPin, HIGH);  // turn the LED on (HIGH is the voltage level)
  Serial.print("HIGH\n");
  delay(1000);                       // wait for a second
  digitalWrite(ledPin, LOW);   // turn the LED off by making the voltage LOW
  Serial.print("LOW\n");
  delay(1000);                       // wait for a second
}
```

Figure 10 – The code used for the above implementation.

The main responsibility of NodeMCU in our project is connecting to Wi-Fi and the MQTT server, and sending respective signals to RF Receiver through the RF Transmitter. Our NodeMCU should be able to both receive and send signals using an external library called "`RCSwitch`". After setting up a transmitter and a receiver for our module using this library, we need different codes to handle the functionality of each part. This will be handled through the "`loop`" function deciding on which procedure to call based on the machine's state. State 1 is the state for learning, state 2 is for transmitting, and the other state is for remaining idle.

```
void loop() {
  checkForResetButton();

  client.loop();

  connected_to_wifi = connectToWifi();

  if (connected_to_wifi && !connected_to_mqtt) {
    Serial.println("Connecting to mqtt");
    connected_to_mqtt = connectToMqtt();
    Serial.println("Connected to mqtt");
  }
  if (!client.connected() && connected_to_mqtt) {
    Serial.println("mqtt disconnected");
    connected_to_mqtt = false;
  }


  if (state == 0) {
    // nothing yet
  } else if (state == 1) {  // learn
    learn();
  } else if (state == 2) {
    transmit();
  } else {
    state = 0;
  }
}
```

Figure 11 - The loop function deciding procedure based on state.

The machine's state is determined by another function "`callback`", which is called whenever a message is received through MQTT.

```
void callback(char *topic, byte *payload, unsigned int length)
{
  Serial.printf("Call back %s %s \n", topic, payload);
  DynamicJsonDocument data(length + 10);
  DeserializationError err = deserializeJson(data, payload);
  if (err)
  {
    Serial.print(("deserializeJson() failed: "));
  }
  if (!data.containsKey("mode") || !data.containsKey("id")) {
    sendError("Invalid json received thus ignored");
  }
  const char* mode = data["mode"];
  int id = data["id"];
  if (strcmp(mode, "learn") == 0) {
    selectedId = id;
    state = 1;
  } else if (strcmp(mode, "transmit") == 0) {
    selectedId = id;
    state = 2;
  } else { // cancel
    state = 0;
    selectedId = 0;
  }
  return;
}
```

Figure 12 – The "callback" function.

RF Module

RF module operates at Radio Frequency. This frequency range varies between 30 kHz & 300 GHz. The RF module we used is a combination of an RF Transmitter and an RF Receiver. The transmitter/receiver (Tx/Rx) pair operate at a frequency of 315 MHz.

Codes

The process of reading from memory and writing to it is implemented using the "`preferences`" library, enabling us to write values of type `int` and `long` in desired addresses, defined by strings. The two functions that make use of this library are "`learn`" and "`transmit`", both shown in the figures below. The "`learn`" function gets needed values to save in NodeMCU through RCSwitch's receiver, and then sends a message if learning was successful. "`transmit`" function reads from the memory using the address created from "`id`", and then transmits the read signal with RCSwitch's transmitter.

```
void learn()
{
  Serial.print("Learning for id = ");
  Serial.println(selectedId);
  if (selectedId < 1 || selectedId > 5) {
    sendError("id must be between 1 and 5");
  } else {
    // do the actual learning
    if (receiver.available()) {
      int value = receiver.getReceivedValue();
      if (value == 0) {
        Serial.print("Unknown encoding");
      } else {
        unsigned long receivedValue = receiver.getReceivedValue();
        unsigned int bitLength = receiver.getReceivedBitlength();
        unsigned int protocol = receiver.getReceivedProtocol();
        Serial.print("Received ");
        Serial.print( receivedValue );
        Serial.print(" / ");
        Serial.print( bitLength );
        Serial.print("bit ");
        Serial.print("Protocol: ");
        Serial.println(protocol);

        // persist in memory
        char key[10];
        sprintf(key, "value%d", selectedId);
        preferences.putULong(key, receivedValue);

        sprintf(key, "bitLength%d", selectedId);
        preferences.putUInt(key, bitLength);

        Serial.println("Learning complete");
        sendMessage("Learned");
        state = 0;
      }
      receiver.resetAvailable();
    }
  }
}
```

Figure 13 – Function that saves signals to memory.

```
void transmit()
{
  Serial.print("Transmitting for id = ");
  Serial.println(selectedId);

  if (selectedId < 1 || selectedId > 5) {
    sendError("id must be between 1 and 5");
  } else {
    // TODO: read from memory
    char key[10];
    sprintf(key, "value%d", selectedId);
    unsigned long value = preferences.getULong(key, 0);
    sprintf(key, "bitLength%d", selectedId);
    unsigned int bitLength = preferences.getUInt(key, 0);

    Serial.print("Read values from memory -> value: ");
    Serial.print(value);
    Serial.print(" , bitLength: ");
    Serial.println(bitLength);

    // TODO: do the actual transmitting

    transmitter.send(value, bitLength);
    Serial.println("Transmitted");
    sendMessage("Sent");
  }
  state = 0;
}
```

Figure 14 – Function that reads signals from memory and then transmits them.

Hardware

Below is the figure of our Transmitter and Receiver.

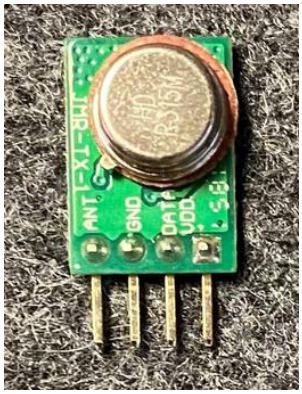

Figure 16 – RF Transmitter.

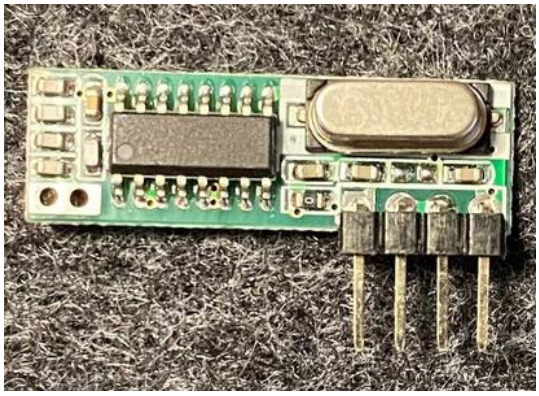

Figure 15 – RF Receiver.

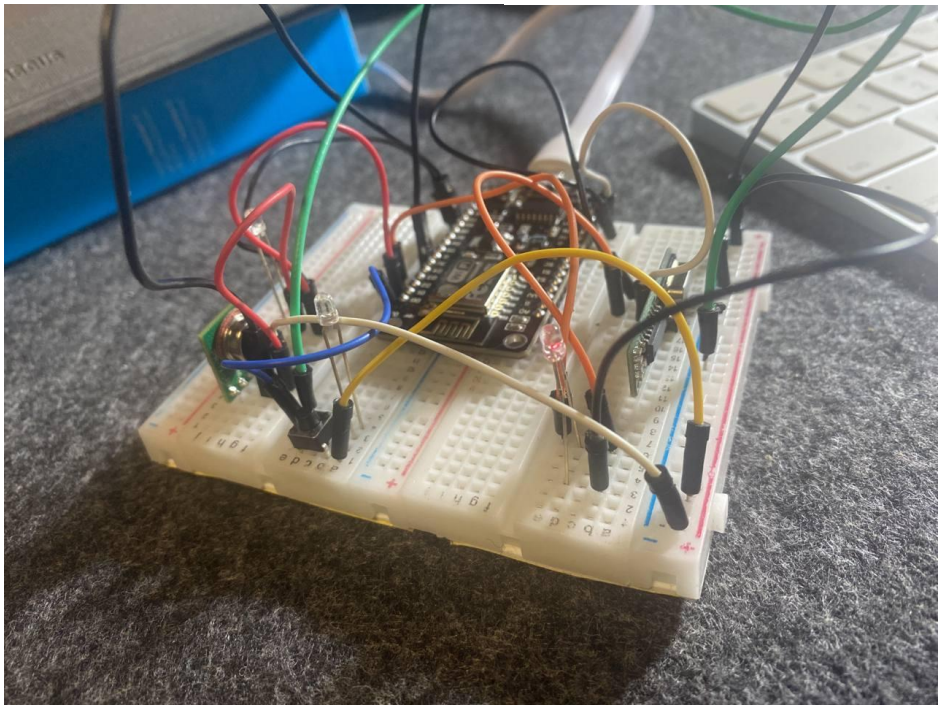

Figure 17 – An overview of the final project.

Simulation

To test the functionality of our design with the RF Transmitter/Receiver, we used the modules implemented in Proteus8 with two Arduino Unos, simulating a simple project of turning on an LED connected to the RF Receiver using the signals sent by the RF Transmitter. The successful implementation of this simulation was a stepping stone for the rest of our project.

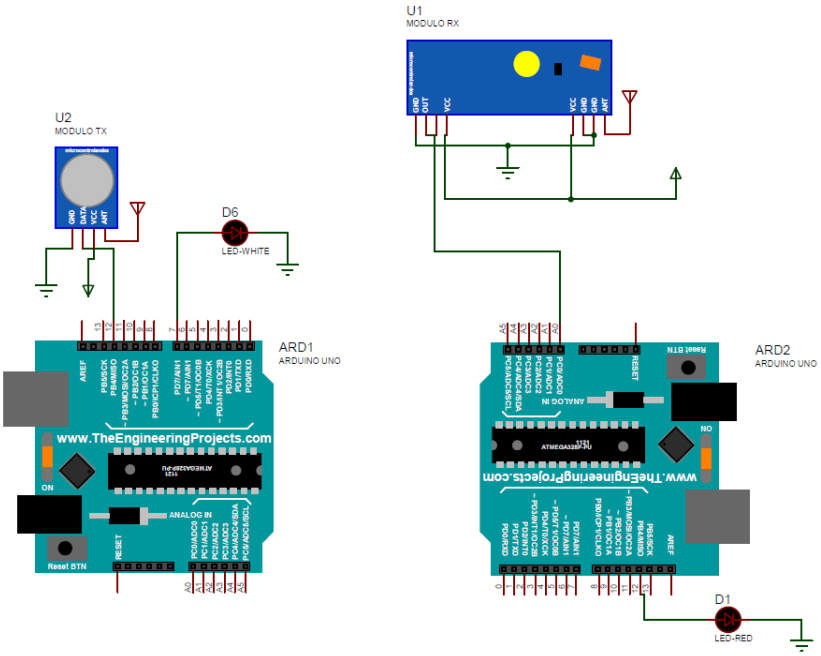

Figure 18 – Simulation in Proteus8, using RF modules and two Arduinos.

Bonus – Using a Key to Open the Door

By using another NodeMCU connected to a key, we implemented the bonus objection of our project. Whenever this key is pushed, a signal is sent to our MQTT broker, which will be forwarded to our implemented remote. The procedure for the remote does not have to change, and the key will work as any other client connected through the IOS application. The functions called after pressing the key are shown below: (the rest of the functions are pretty similar to the previous parts explained above)

```
void checkForButton()
{
  if (digitalRead(theKeyPin) == HIGH)
  {
    sendMessage();
  }
}
```

Figure 19 – Checking whether or not the key is pressed.

```
void sendMessage() {
  DynamicJsonDocument doc(2048);
  doc["mode"] = "transmit";
  doc["id"] = selectedId;
  char buf[2048];
  serializeJson(doc, buf);
  client.publish(PUB_TOPIC, buf);
  Serial.println("Sent message");
}
```

Figure 20 – Sending message after key press.

A video showing the final version of our project and its functionality is available here.

Resources

[1] https://mqtt.org/

[2] https://www.emqx.io/

[3] https://www.make-it.ca/nodemcu-details-specifications/

[4] https://robu.in/what-is-rf-transmitter-and-receiver/

[5] https://maker.pro/arduino/projects/how-to-simulate-arduino-projects-using-proteus