DOCUMENTATION

of

DFPL Project001

December 2019

Contents

List of Figures

List of Tables

Chapter 1

Prerequisite Concepts

1.1 Internet of Things

Internet of Things (IoT) [1] is an ecosystem of devices that are connected to the internet. The ‘thing’ can be anything from a heart monitor, an automobile, temperature sensor to a refrigerator. It includes any object that can transmit or receive data either directly or indirectly to the internet without manual assistance. The advent of wireless technology has lead to a rapid decrease in costs of such a system and increased the viability of a large deployment of IoT devices exponentially.

The devices are managed using an IoT platform which is a multi-layer technology that enables straightforward provisioning and management of connected devices. This has lead to IoT being widely used in connecting devices and collecting data information. The system is used to register their sensors, manage streams of data and handle configuration updates. Using a cloud-based service for the same greatly speeds up the development of applications and takes care of scalability and cross-device compatibility as well [2].

This connects your diverse hardware using a range of enterprise-grade connectivity options to huge data processing solutions, opening up a plethora of applications ranging from data collection to drone delivery networks to precision farming [3].

1.2 Wireless Sensor Network

A Wireless sensor network (WSN) refers to a group of spatially dispersed connected sensors that can be used for monitoring and recording the environmental conditions and creating a network to route that data to a central location. WSNs in precision farming increase efficiency and profitability by reducing networking costs and deployment complexity [4]. This gives real-time access to environmental information remotely which can be used for storage, data analytics, and make resource decisions. This contrasts with the traditional agricultural methods in which decisions were taken based on tradition and habit.

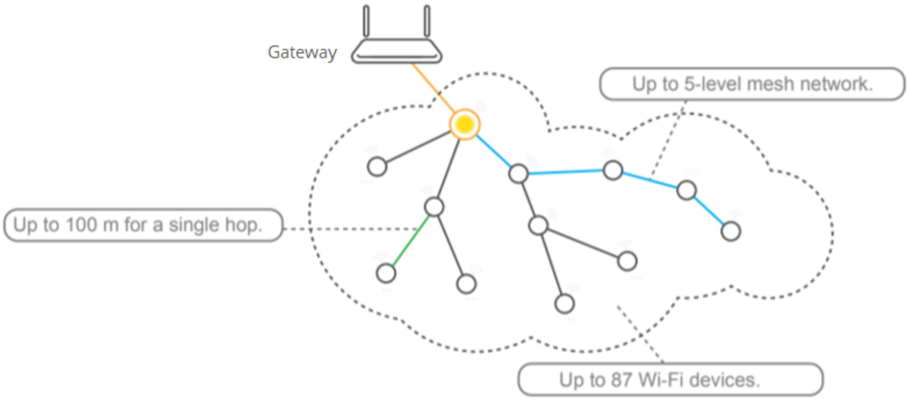

Figure 1.1: ESP8266 Mesh Network

WSNs make a large scale deployment of sensors economically viable for the farming domain [5]. It supports many operations such as irrigation, fertilizer use, soil monitoring and intruder detection [6]. Its integration has resulted in a plethora of applications such as remote healthcare, water control, precision agriculture, smart cities, and wildlife monitoring [7].

1.3 Gateway

An IoT gateway is a device/software that serves as the bridge between the WSN and the cloud side. It is responsible for device management, data processing, and routing [8].
In a cloud-based network architecture, these gateways act as edge nodes, reducing the amount of processing power required on the cloud end. As seen in Figure 1.2, this reduces both the cost and the complexity of the network.
It performs several tasks on their behalf, such as: [8]

- Communicating with IoT Platform;
- Connecting to the internet when the device can't directly connect itself, such as a ZigBee or Bluetooth device;
- Providing secure authentication when the device can't send its credentials, or when you want to add a layer of security by using the credentials of both the device and the gateway;

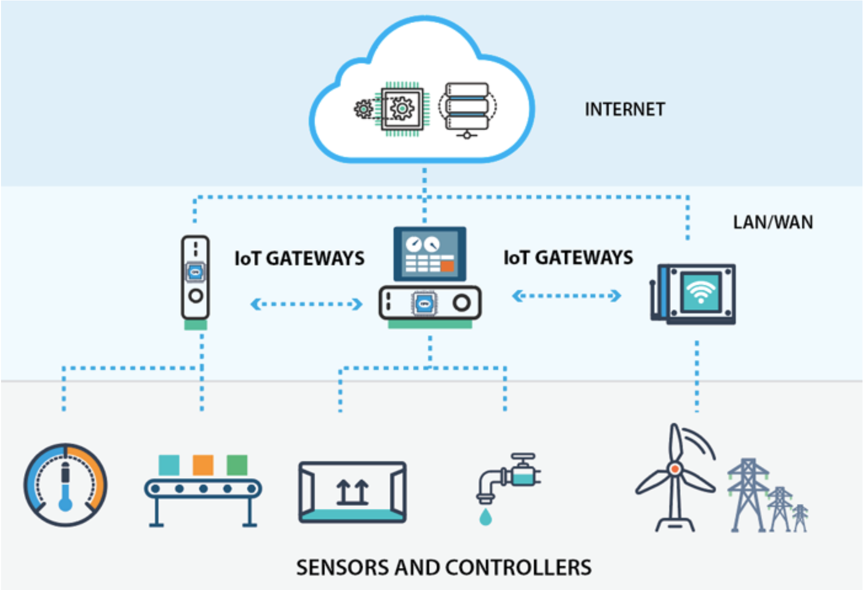

Figure 1.2: Role of an IoT Gateway [9]

- Publishing telemetry events, device management, getting configuration data, or setting device state;
- Storing and processing data, logs and telemetry; and
- Translating between different protocols.

Chapter 2

Solution Architecture

2.1 Requirements and Design Consideration

The requirements for this project are clear and fixed. The functional requirements are as follows:

1. The platform should scale easily with the number of devices and sensing parameters;
2. WSN should be inexpensive and easy to deploy;
3. The platform should have archival cloud storage with data analytics capability;
4. The cloud service must be reliable and always on;
5. Devices must be authenticated before sending telemetry data;
6. Devices should be managed and reconfigured via over-the-air (OTA) updates;
7. Data should be accessible in real-time by the user via MQTT; and
8. The telemetry parameters required for each sensing node are given in Table 2.1.

Based on the functional requirement, the proposed Software Design Considerations are given in Table 2.2.

Service Type	Telemetry Parameters
Publish	Soil Temperature Soil Moisture Timestamp NodeID FarmID
Subscribe	Configuration State Commands

Table 2.1: Device Telemetry Parameters

	Requirement	Design Consideration
1	Scalable with the number of devices and sensing parameters	An easily scalable WSN to be used.
2	WSN should be inexpensive and easy to deploy	WSN shouldn't be based on proprietary software/hardware; preferably consumer hardware
3	Cloud Platform must have archival cloud storage with data analytics capability	Utilizing Google BigQuery for its high throughput and big data analytics capability
4	Cloud service must be reliable and always on	Google Cloud Platform is reliable and has high availability
5	Devices must be authenticated before sending telemetry data	Sensing nodes will be authenticated using an intermediary device keeping the cost down
6	Devices should be managed and reconfigured via over-the-air (OTA) updates	GCP IoT Core provides device management and configuration over the air
7	Data should be accessible in real-time by the user via MQTT	Real-time data will be made available through an MQTT broker hosted by Google Cloud Pub/Sub

Table 2.2: Functional Requirements and Design Considerations

2.2 Network Architecture

The proposed network architecture (depicted in Figure 2.1) is composed of 3 layers: sensor, gateway, and cloud back-end. In this section, we will discuss these three layers and their implementation.

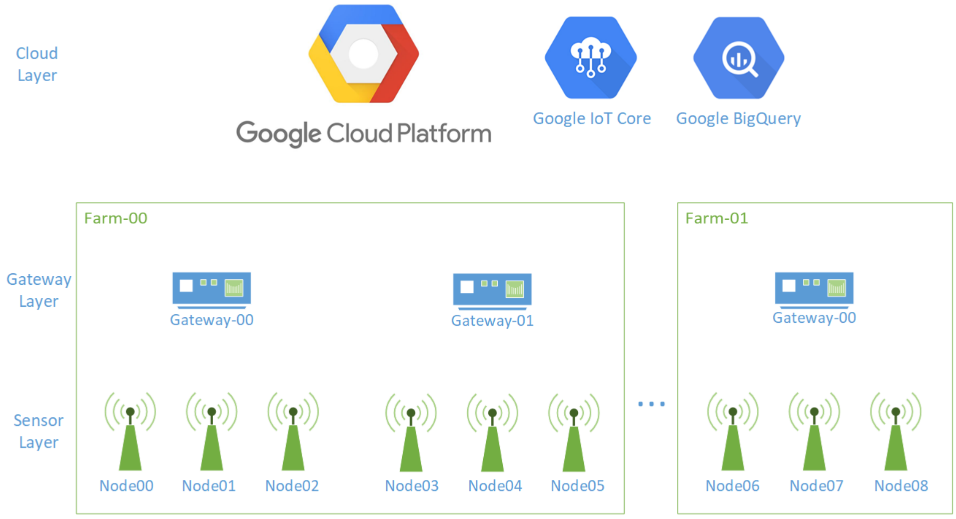

Figure 2.1: Proposed 3-layer architecture

2.2.1 Sensor Layer

The sensor layer is comprised of the actual sensing hardware. Each device or node currently comprises of just 3 modules: an IoT capable micro-controller(ESP8266), the environmental sensors (Soil Moisture and Temperature), and a 3.2v battery (LiFePo4). The functionally modular design allows for easy upgradability and repair. The node design is given in Figure 2.2.

The ESP8266 microcontroller is a low-cost, low-power microcontroller with inbuilt WiFi capability and multiple GPIOs for communication purposes (Refer to Table 2.3). The micro-controller is responsible for collecting the data from the sensors and sending it to the gateway through the mesh. The micro-controller is powered by a 3.3v power source and uses 80-90mA during load and 20µA during deep sleep. Using a 1-3 hour duty cycle, the power consumption of the micro-controller can be reduced to less than 1mA per hour. Using a 3000mah battery, this can last anywhere from 150-215 days depending on external factors.

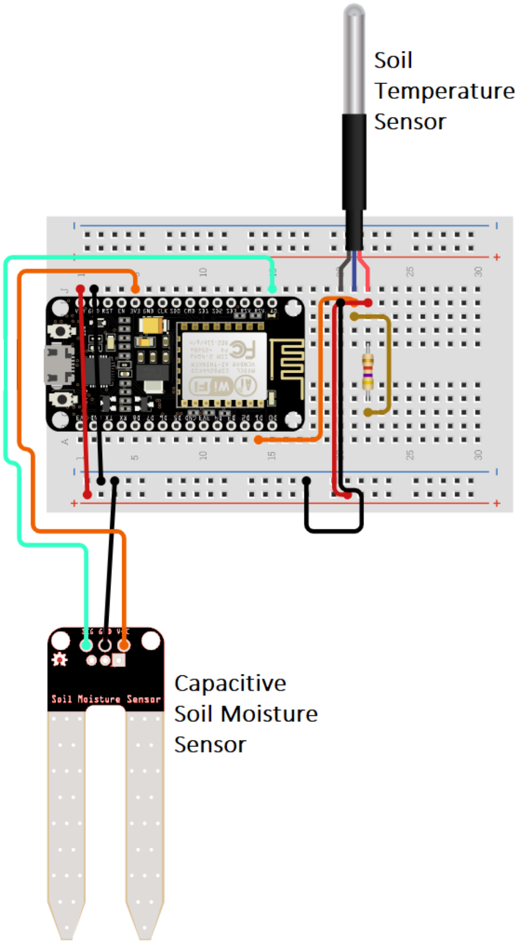

Figure 2.2: Node Design Schematics

Parameters	Specification
Processor	TenSilica L106 80MHz 32bit
SRAM	160 kbytes
Flash Storage	SPI Flash, up to 16 MBytes
GPIO pins	17
ADC pins	1 pin, 10 bit
WiFi	2.4GHz, 802.11 b/g/n
Operating Voltage	3-3.6 Volts

Table 2.3: ESP8266 (Sensor Node) Specifications

Sensor	**Model**
Soil Temperature	DS18B20
Soil Moisture (Capacitive)	RKI-3225

Table 2.4: Used Sensor Models

2.2.2 Gateway Layer

The gateway layer comprises aggregating devices that connect to WSN networks, collect the data, authenticate the devices, and then relay the data to the cloud. During each duty cycle, the gateway will wake up and perform the following:

1. Connect to the WSN using wifi;
2. Authenticate active sensor nodes using a JSON Web Token (JWT);
3. Collect sensor reading and calculate any derived measurements;
4. Relay to Cloud MQTT broker and the BigQuery database for archival storage with proper timestamps; and
5. Receive any configuration updates and relay them to the respective node in the WSN.

The gateway is implemented using Raspberry Pi 4 (4GB) micro-controller. Being equipped with a 1.5GHz quad-core CPU and 4 GB RAM, it is more than capable enough to handle multiple data streams and adds to the scalability of the design. These devices provide sufficient edge processing, removing any requirement for cloud-based processing power. The inbuilt WiFi(2.4 GHz and 5.0 GHz), Bluetooth 5.0, and BLE modules can be used for connecting to both Wifi and BLE based WSNs. A 4G USB dongle is used to provide internet connectivity in remote areas, but an ethernet interface can also be used for the same.

2.2.3 Cloud Layer

The cloud layer is responsible for data ingestion, device management through the gateway, and providing archival storage and data analytics. The cloud architecture is described in Figure 2.3.

The Google Cloud Platform (GCP) is used for the same as Google Cloud IoT core platform has the functionality to support data ingestion from a large number of globally distributed devices. The platform also has integrated services that manage authentication and device management. Google

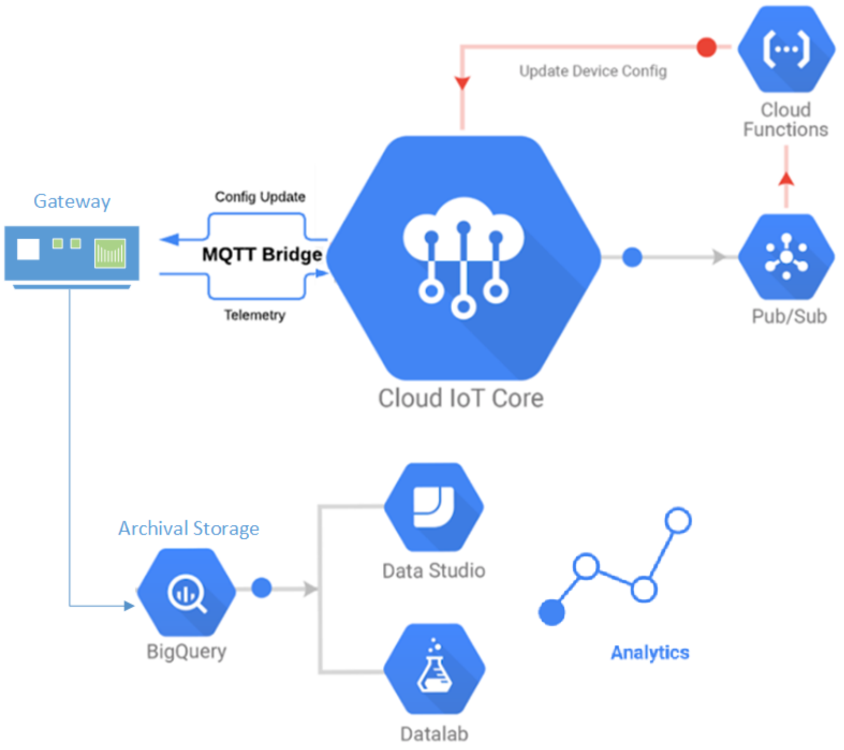

Figure 2.3: GCP Cloud Architecture

BigQuery is used for archival storage and data analytics as it supports direct streams of data and has high throughput [10, 11].

A direct connection between an ESP8266-based node and Google Cloud IoT Core is possible but since many inexpensive boards lack the features required to make a secure connection, the WSN is kept isolated. The gateway layer handles the authentication process for the nodes through JSON Web Token (JWT) authentication. Each node must be authenticated via the gateway it is bound to before it can send any telemetry data.

Chapter 3

Solution Stack

3.1 Hardware Stack

The two main factors that decided the hardware stack are cost and functionality. The node needs to be as inexpensive as possible as to keep the costs down when scaling, and the gateway needs to be powerful enough to allow easy scalability and upgradability.

The 2 microcontrollers that are a great fit to serve as the brain of the node, ESP8266 and ESP32. Both are perfectly suited and the software stack is compatible with both of them. To keep the development costs down, the ESP8266 was used. The complete bill of materials is given in Table 3.1.

Device	**Components**	**Cost**	
9 x Nodes			₹ 1,050
	ESP8266	₹ 364	
	Soil Moisture Sensor	₹ 177	
	DS18B20 Soil Temperature Sensor	₹ 110	
	3000mah LiFePo4 Battery	₹ 299	
	Housing	₹ 100	
1 x Gateway			₹ 5,272
	Raspberry Pi 4 (4GB)	₹ 3,994	
	16GB Storage	₹ 279	
	4G Dongle	₹ 999	
Total:			₹ 14,722

Table 3.1: Bill of Materials

3.2 Software Stack

3.2.1 Node

The ESP8266 node is programmed using PlatformIO [12], an open-source, cross-platform IDE. This allows for fast debugging, code execution, and testing on any platform that can run python. The code is written in C++ (refer to Appendix 8.1).

The mesh is built and configured using painlessMesh [13] module, an open-source library that takes care of the particulars of creating a simple ad-hoc mesh network which required no central controller. It is compatible with both the ESP8266 and the ESP32. Each node created a wifi network with the same SSID but different BSSID. After the mesh is initialized and configured, only one SSID is publicly visible, reducing any network clutter. The main loop logic is defined as follows:

1. Initialize the mesh and wait for the gateway to connect (until max wait time);
2. When connected, request for authentication, and sense the parameters;
3. When authenticated, send the payload with the available parameters;
4. Receive any configuration updates, or commands from the gateway and perform the necessary actions; and
5. Go to deep sleep until the next wakeup cycle.

3.2.2 Gateway

The gateway powered by the Raspberry Pi 4 hardware is programmed in python, allowing for cross-platform compatibility and support. The python script works alongside a modified version painlessMeshBoost library [14] which creates a bridge between the WSN mesh and the gateway.

The setup instructions are as simple as running the python script with the appropriate arguments. The main loop logic is defined as follows:

1. Actively looks for the WSN and connect when available;
2. When connected, using the painlessMeshBoost bridge receive the telemetry data from the Sensor nodes and act accordingly;
3. Relay the telemetry data with appropriate timestamps to Google Cloud IoT Core, and BigQuery database for archival storage;
4. Relay any configuration updates and commands to the specific nodes using the bridge; and
5. Sleep until the next wakeup cycle.

Chapter 4

Full Deployment Process

The link for the full code base is: `https://github.com/Raghav-intrigue/dfpl-project001`
To download the full solution:

```
git clone --recurse-submodules \
  https://github.com/Raghav-intrigue/dfpl-project001
cd dfpl-project001
```

Then, the deployment from scratch requires:

4.0.1 Initial cloud setup

This will be done only once in the projects lifetime. Follow the instructions in Cloud setup from scratch

4.0.2 Device setup and registration

Each device (node and gateway) has a unique ID, this needs to be registered on the cloud. This ID can be any arbitrary string (ensure consistency in the naming schema for convenience). This will be done once for each device in it's lifetime.

1. Setup Cloud (open Cloud Setup):
    - Register Gateway (note down the gatewayID)
    - Register Nodes and Bind the nodes to the gateway (note the nodeIDs)
2. Flash each node with the registered NodeIDs. See Node Setup
3. Setup Gateway (software installation). See section Gateway Setup

4.0.3 Field Deployment

1. Turn on the gateway (Raspberry Pi)
    - Connect the given USB C power adapter to the raspberry pi.
    - Connect the given 4G dongle to raspberry pi via usb
    - Turn on the raspberry pi
2. Place the nodes in the field
3. Turn them on

Chapter 5

Next Steps

The platform developed supports and is designed for various decision driven applications, which would include actuators and valves for controlling the amount of farm inputs. In the process of making the architecture industry ready, the following steps are needed:

- Designing a front end for platform management and deployment;
- Designing industry-ready sensors for robust hardware and more reliable measurement;
- Improving the mesh architecture for more stability; and
- Adding field imagery and drone footage as sensory input.

Chapter 6

Code:

6.1 Code for Sensor Node

```
#include <Arduino.h>
#include <painlessMesh.h>

#include <ESP8266WiFi.h>
#include <OneWire.h>
#include <DallasTemperature.h>

/**
 * @brief The node ID given to the node
 * Keep it unique for each node
 * Change this before flashing
 */
#define NODE_ID ”espmesh−00”

// GPIO where the DS18B20 data pin is connected
const int oneWireBus = 4;

// Setup a oneWire instance to communicate with any OneWire
    devices
OneWire oneWire(oneWireBus);
// Pass our oneWire reference to Dallas Temperature sensor
DallasTemperature sensors(&oneWire);

#define LED 2 // GPIO number of connected LED, ON ESP−12 IS
    GPIO2

#define MESH_SSID ”whateverYouLike”
#define MESH_PASSWORD ”somethingSneaky”
#define MESH_PORT 5555

//number of seconds between each reading
#define messageDelayTime 10

//number of seconds between each debug message
#define debugFreq 3
```

```
//number of seconds before the node goes to sleep
#define sleepFreq 60

//number of micro-seconds the node remains in sleep
#define sleepDuration 60e6

//number used to identify the gateway (keep constant)
#define GATEWAY_ID 696969

//constants
#define TYPE_DEBUG "debug" //Represents debug messages
#define TYPE_ERROR "error" //Represents error messages
#define TYPE_SYSTEM "system" //Represents system messages
#define TYPE_DATA "data" //Represents telemetry messages

// Prototypes
void sendTelemetryPayload();
void receivedCallback(uint32_t from, String &msg);
void newConnectionCallback(uint32_t nodeId);
void changedConnectionCallback();
void nodeTimeAdjustedCallback(int32_t offset);
void delayReceivedCallback(uint32_t from, int32_t delay);

Scheduler userScheduler; //to control your personal task
painlessMesh mesh;

bool calc_delay = false;
SimpleList<uint32_t> nodes;

void sendTelemetryPayload();
void printDebugOutput();
void initSleep();

/**
 * @brief Used to schedule callbacks after a given delay
 * delay() function cannot be used as it will block the mesh
     functions
 *
 */
Task taskSendData(TASK_SECOND *messageDelayTime, TASK_FOREVER, &
    sendTelemetryPayload);
Task taskDebugOutput(TASK_SECOND *debugFreq, TASK_FOREVER, &
    printDebugOutput);
Task initiateDeepSleep(TASK_SECOND *sleepFreq, TASK_FOREVER, &
    initSleep);

//to avoid the first sleep cycle
bool first = true;

/**
 * @brief Initialise painlessMesh node and set up callbacks
 *
 */
```

```
void setup()
{
  Serial.begin(115200);

  first = true;

  mesh.setDebugMsgTypes(ERROR | DEBUG); // set before init() so
      that you can see error messages

  mesh.init(MESH_SSID, MESH_PASSWORD, &userScheduler, MESH_PORT)
      ;
  mesh.onReceive(&receivedCallback);
  mesh.onNewConnection(&newConnectionCallback);
  mesh.onChangedConnections(&changedConnectionCallback);
  mesh.onNodeTimeAdjusted(&nodeTimeAdjustedCallback);
  mesh.onNodeDelayReceived(&delayReceivedCallback);

  //schedule and enable tasks
  userScheduler.addTask(taskSendData);
  userScheduler.addTask(taskDebugOutput);
  userScheduler.addTask(initiateDeepSleep);
  taskDebugOutput.enable();
  taskSendData.enable();
  initiateDeepSleep.enable();
}

/**
 * @brief Returns the json string with the given payload
 *
 * @param type: can be TYPE_DATA, TYPE_ERROR, TYPE_SYSTEM,
    TYPE_DEBUG
 * @param data: the payload to send
 *
 * @return String
 */
String make_json(char type[], char data[])
{
  DynamicJsonDocument doc(1024);

  doc[”nodeID”] = NODE_ID;
  doc[”type”] = type;
  doc[”data”] = data;

  String output;
  serializeJson(doc, output);
  Serial.print(”Sending: ”);
  serializeJson(doc, Serial);
  Serial.println(””);

  //escape ” character from json
  String ret;
  for (unsigned int i = 0; i < output.length(); i++)
  {
    if (output[i] == '”')
    {
      ret += "\\\"";
    }
    else
      ret += output[i];
  }
  return ret;
}
String make_json(char type[], JsonObject data)
{
  DynamicJsonDocument doc(1024);

  doc["nodeID"] = NODE_ID;
  doc["type"] = type;
  doc["data"] = data;

  String output;
  serializeJson(doc, output);

  Serial.print("Sending: ");
  serializeJson(doc, Serial);
  Serial.println("");

  //escape " character from json
  String ret;
  for (unsigned int i = 0; i < output.length(); i++)
  {
    if (output[i] == '"')
    {
      ret += "\\\"";
    }
    else
      ret += output[i];
  }
  return ret;
}

/**
 * @brief Get the readings as a json string payload
 *
 * @return String
 */
String get_reading()
{
  //analog value from the moisture sensor (range: 0-1023)
  int mois = analogRead(0);

  //extract moisture value from the analof reading
  float RH = (750.0 - (float)mois) / 7.50;

  //get temperature reading
  sensors.requestTemperatures();
  float temperatureC = sensors.getTempCByIndex(0);
  //create json payload
  DynamicJsonDocument doc(1024);
  doc["temperature"] = temperatureC;
  doc["moisture"] = RH;
  JsonObject data = doc.as<JsonObject>();

  return make_json(TYPE_DATA, data);
}

/**
 * @brief notifies the gateway to authenticate this node
 *
 */
void setupCloud()
{
  mesh.sendSingle(GATEWAY_ID, make_json(TYPE_SYSTEM, "init"));
}

/**
 * @brief main repeating loop
 *
 */
void loop()
{
  mesh.update();
}

/**
 * @brief Sends the telemetry payload to the gateway with the
     sensor readings.
 *
 */
void sendTelemetryPayload()
{
  mesh.sendSingle(GATEWAY_ID, get_reading());
  taskSendData.setInterval(TASK_SECOND * messageDelayTime);
}

/**
 * @brief Callback for when a message is received
 *
 * @param from: node_id of the sender
 * @param msg: message received
 */
void receivedCallback(uint32_t from, String &msg)
{

  Serial.printf("Received from %u msg=%s\n", from, msg.c_str());

  //authenticate if the gateway has connected to the mesh
  if (msg.compareTo("gatewayConnected") == 0)
  {
    Serial.printf("%s: Gateway connected to %d", NODE_ID, from);
    setupCloud();
  }
}

/**
 * @brief Callback for when a new node connects to this node
 *
 * @param nodeId: node id of the connected device
 */
void newConnectionCallback(uint32_t nodeId)
{
  if (nodeId == GATEWAY_ID)
  {
    setupCloud();
    mesh.sendBroadcast("gatewayConnected");
  }
  Serial.printf("New Connection, nodeId = %u\n", nodeId);
  Serial.printf("New Connection, %s\n", mesh.subConnectionJson(
      true).c_str());
}

/**
/**
 * @brief Callback for when the mesh connections are changed
 *
 */
void changedConnectionCallback()
{
  Serial.printf("Changed connections\n");

  //prints the connection list
  nodes = mesh.getNodeList();

  Serial.printf("Num nodes: %d\n", nodes.size());
  Serial.printf("Connection list:");

  SimpleList<uint32_t>::iterator node = nodes.begin();
  while (node != nodes.end())
  {
    Serial.printf(" %u", *node);
    node++;
  }
  Serial.println();
  calc_delay = true;
}

/**
 * @brief Callback for when the node time is adjusted
 *
 * @param offset
 */
void nodeTimeAdjustedCallback(int32_t offset)
{
  Serial.printf("Adjusted time %u. Offset = %d\n", mesh.
      getNodeTime(), offset);
```

```
}

/**
 * @brief Callback for when the delay is measured between nodes
 *
 * @param from: node_id of the first node
 * @param delay: delay in micro-seconds
 */
void delayReceivedCallback(uint32_t from, int32_t delay)
{
  Serial.printf("Delay to node %u is %d us\n", from, delay);
}

/**
 * @brief prints debug output
 *
 */
void printDebugOutput()
{
  Serial.printf("Tick - Time:%u ID:0 Mesh Size: %d\n", mesh.
      getNodeTime(), nodes.size() + 1);
}

/**
 * @brief function that initiates DeepSleep
 *
 */
void initSleep()
{
  // Todo: Do any necessary housekeeping
  if (first)
  {
    first = false;
    return;
  }
  Serial.println("Going to sleep");
  mesh.stop();
  ESP.deepSleep(sleepDuration);
}
```

6.2 Code for the gateway

```
# region imports

import argparse
import datetime
import json
import os
import socket
import ssl
import time
from time import ctime
from time import sleep
```

```
import pytz
import sys

import logging

# persistance
import pickle

# mqtt
import jwt
import paho.mqtt.client as mqtt

# for running shell commands
import platform
import subprocess
import shlex

# for big query
from google.cloud import bigquery
from google.cloud.exceptions import NotFound

# import variables and constants
from dfpl_gateway import *

# endregion

# defines the print output verbosity
READING_LEVEL = 5

# class instances
logger = {}
process = {}


# cloud setup variables
# change these according to your cloud setup
class SetupVariables:
    project_id = 'dfpl-project-001'
    cloud_region = 'asia-east1'
    registry_id = 'dfpl-temp'
    gateway_id = 'test-gateway'
    algorithm = 'RS256'
    mqtt_bridge_hostname = 'mqtt.googleapis.com'
    mqtt_bridge_port = 8883
    jwt_expires_minutes = 1200
    path_to_bqcreds = ''


setupVars = SetupVariables()

# stores all information related to the gateway's current state
class GatewayState:
    # This is the topic that the device will receive
        configuration updates on.
```

```
mqtt_config_topic = ''

# This is the configuration, updated from cloud.
# Used to store the variables that define the normal
    operation of the gateway
# farmID
# datasetID
config = {"farmID": "farm_001", "datasetID": "rawSensorData"
    }

# Host the gateway will connect to
mqtt_bridge_hostname = ''
mqtt_bridge_port = 8883

# For all PUBLISH messages which are waiting for PUBACK. The
     key is 'mid'
# returned by publish().
pending_responses = {}

# For all attach messages which are waiting for PUBACK.
# Subscriptions are executed after ACK is received
pending_attach = {}

# For all SUBSCRIBE messages which are waiting for SUBACK.
    The key is
# 'mid'.
pending_subscribes = {}

# Numeric nodeId_nums assigned by mesh
nodeID_num_for = {}

# for all SUBSCRIPTIONS. The key is subscription topic.
subscriptions = {}

# Indicates if MQTT client is connected or not
connected = False

def __init__(self):
    try:
        file = open('saved.config', 'rb')
        self.config = pickle.load(file)
        logger.debug('Loaded␣saved␣config:␣{}'.format(self.
            config))
    except FileNotFoundError:
        logger.warning('Saved␣config␣not␣found')

def update_config(self, conf):
    self.config = conf
    file = open('saved.config', 'wb')
    pickle.dump(self.config, file)
    file.close()
    logger.debug('Updated␣config:␣{}'.format(self.config))
```

```
gateway_state = {}

# handles archival storage to the BigQuery database
class BigQueryHandler:

    def __init__(self):
        global setupVars
        global gateway_state

        # Instantiates a client
        self.bigquery_client = bigquery.Client.
            from_service_account_json(
            setupVars.path_to_bqcreds)

        try:
            self.datasetID = gateway_state.config['datasetID']
        except KeyError:
            self.datasetID = 'rawSensorData'

        self.get_dataset()

    # Prepares a reference to the dataset
    def get_dataset(self):
        self.dataset_ref = self.bigquery_client.dataset(self.
            datasetID)
        try:
            self.bigquery_client.get_dataset(self.dataset_ref)
        except NotFound:
            # Create Dataset
            dataset = bigquery.Dataset(self.dataset_ref)
            dataset = self.bigquery_client.create_dataset(
                dataset)
            logger.info('Dataset␣{}␣created.'.format(dataset.
                dataset_id))

    # Prepares a reference to the specific table
    def get_table(self):
        global gateway_state
        farmID = gateway_state.config['farmID']

        # Prepares a reference to the table
        self.table_ref = self.dataset_ref.table(farmID+'
            _readings')

        # timestamp:TIMESTAMP, nodeID:STRING, temperature:FLOAT,
            moisture:FLOAT
        # {"temperature": 24.875, "moisture": 7.866667, "
            timestamp": "11-11-2019, 19:14:57", "nodeID": "
            espmesh-00"}
        try:
            self.bigquery_client.get_table(self.table_ref)
        except NotFound:
            schema = [
                bigquery.SchemaField(
                    'timestamp', 'TIMESTAMP', mode='REQUIRED'),
                bigquery.SchemaField('nodeID', 'STRING', mode='
                    REQUIRED'),
                bigquery.SchemaField('temperature', 'FLOAT',
                    mode='REQUIRED'),
                bigquery.SchemaField('moisture', 'FLOAT', mode='
                    REQUIRED'),
            ]
            table = bigquery.Table(self.table_ref, schema=schema
                )
            table = self.bigquery_client.create_table(table)
            logger.warning('table␣{}␣created.'.format(table.
                table_id))

    # streams the telemetry to the BQ table
    def addToBQ(self, data):
        self.get_table()

        table = self.bigquery_client.get_table(self.table_ref)
            # API call

        rows_to_insert = [data]
        errors = self.bigquery_client.insert_rows(
            table, rows_to_insert)  # API request
        return errors


bqHandler = {}


# region IoT Core Setup

# Creates a jwt token for authenticating to Google Cloud IoT
    Core
def create_jwt(project_id, private_key_file, algorithm,
    jwt_expires_minutes):
    """Creates a JWT (https://jwt.io) to establish an MQTT
        connection.
            Args:
             project_id: The cloud project ID this device
                 belongs to
             private_key_file: A path to a file containing
                 either an RSA256 or
                             ES256 private key.
             algorithm: Encryption algorithm to use. Either '
                 RS256' or 'ES256'
             jwt_expires_minutes: The time in minutes before the
                 JWT expires.
            Returns:
                An MQTT generated from the given project_id and
                    private key,
                which expires in 20 minutes. After 20 minutes,
                    your client will
                be disconnected, and a new JWT will have to be
```

```
                    generated.
            Raises:
                ValueError: If the private_key_file does not
                    contain a known
                key.
            """

    token = {
        # The time that the token was issued at
        'iat': datetime.datetime.utcnow(),
        # The time the token expires.
        'exp': (
            datetime.datetime.utcnow() +
            datetime.timedelta(minutes=jwt_expires_minutes)),
        # The audience field should always be set to the GCP
            project id.
        'aud': project_id
    }

    # Read the private key file.
    with open(private_key_file, 'r') as f:
        private_key = f.read()

    logger.info('Creating␣JWT␣using␣{}␣from␣private␣key␣file␣{}'
        .format(
        algorithm, private_key_file))

    return jwt.encode(token, private_key, algorithm=algorithm)

# Creates the MQTT client for Cloud IoT Core


def get_client(
        project_id, cloud_region, registry_id, gateway_id,
            private_key_file,
        algorithm, ca_certs, mqtt_bridge_hostname,
            mqtt_bridge_port,
        jwt_expires_minutes):

    # The client_id is a unique string that
    # identifies this device. For Google Cloud IoT Core, it must
         be in the
    # format below
    client_template = 'projects/{}/locations/{}/registries/{}/
        devices/{}'
    client_id = client_template.format(
        project_id, cloud_region, registry_id, gateway_id)
    client = mqtt.Client(client_id)

    # With Google Cloud IoT Core, the username field is ignored,
         and the
    # password field is used to transmit a JWT to authorize the
        device.
    client.username_pw_set(
        username='unused',
        password=create_jwt(
            project_id, private_key_file, algorithm,
            jwt_expires_minutes))

    # Enable SSL/TLS support.
    client.tls_set(ca_certs=ca_certs, tls_version=ssl.
       PROTOCOL_TLSv1_2)

    # Register message callbacks. https://eclipse.org/paho/
        clients/python/docs/
    # describes additional callbacks that Paho supports.

    client.on_connect = on_connect
    client.on_publish = on_publish
    client.on_disconnect = on_disconnect
    client.on_message = on_message
    client.on_subscribe = on_subscribe

    # Connect to the Google MQTT bridge.
    client.connect(mqtt_bridge_hostname, mqtt_bridge_port)

    return client

# endregion


# region MQTT Callbacks

# Convert a Paho error to a human readable string.
def error_str(rc):
    return '{}:␣{}'.format(rc, mqtt.error_string(rc))


# Paho callback for when a device connects.
def on_connect(client, unused_userdata, unused_flags, rc):
    logger.debug('on_connect' + mqtt.connack_string(rc))

    gateway_state.connected = True

    # Subscribe to the config topic.
    client.subscribe(gateway_state.mqtt_config_topic, qos=1)


# Paho callback for when a device disconnects
def on_disconnect(client, unused_userdata, rc):
    logger.info('on_disconnect', error_str(rc))
    gateway_state.connected = False

    # re-connect
    # NOTE: should implement back-off here, but it's a tutorial
    client.connect(
        gateway_state.mqtt_bridge_hostname, gateway_state.
            mqtt_bridge_port)
```

```
# Paho callback when a message is successfully sent to the
    broker
def on_publish(client, userdata, mid):
    logger.debug('PUBACK␣received,␣userdata␣{},␣mid␣{}'.format(
        userdata, mid))

    try:
        # If a PUBACK for attach is received for a Node, it is
            subscribed for updates
        client_addr, message = gateway_state.pending_attach.pop(
            mid)
        logger.info(
            'Device␣{}␣is␣attached.␣Subscribing␣it␣for␣updates.'
                .format(client_addr))
        sub_for_updates(client, client_addr)
        logger.debug('Pending␣attaches␣count␣{}'.format(
            len(gateway_state.pending_attach)))

    except KeyError:
        logger.debug('mid:␣{}.␣Not␣a␣attach␣ack'.format(mid))
        try:
            client_addr, message = gateway_state.
                pending_responses.pop(mid)
            sendTo(client_addr, payload.decode(”utf−8”))
            logger.debug('Pending␣response␣count␣{}'.format(
                len(gateway_state.pending_responses)))
        except KeyError:
            logger.error('Unable␣to␣find␣mid␣{}'.format(mid))


# Paho callback for a SUBACK
def on_subscribe(unused_client, unused_userdata, mid,
    granted_qos):
    logger.info('on_subscribe:␣mid␣{},␣qos␣{granted_qos}'.format
        (mid))
    try:
        client_addr, response = gateway_state.pending_subscribes
            [mid]
        sendTo(client_addr, response)
    except KeyError:
        logger.warning('Unable␣to␣find␣mid:␣{}'.format(mid))


# Callback when the device receives a message on a subscription
# Redirects the message according to the subscription list
def on_message(unused_client, unused_userdata, message):

    global gateway_state

    payload = message.payload
    qos = message.qos
    logger.info('Received␣message␣\'{}\'␣on␣topic␣\'{}\'␣with␣
            Qos␣{}'.format(
            payload.decode("utf-8"), message.topic, qos))

    # Configuration topic for the gateway
    if message.topic == gateway_state.mqtt_config_topic:
        handle_config_update(payload)
        return

    # Configuration topics for the attached nodes
    try:
        client_addr = gateway_state.subscriptions[message.topic]
        sendTo(client_addr, payload.decode("utf-8"))
        logger.info('Sent␣message␣to␣device␣{}'.format(
            client_addr))
    except KeyError:
        logger.warning('Nobody␣subscribes␣to␣topic␣{}'.format(
            message.topic))


# endregion


# region Event Handlers

# subscribes the NodeID to configuration updates and commands
def sub_for_updates(client, nodeID):
    template = '{{␣"device":␣"{}",␣"command":␣"{}",␣"status"␣:␣"
        ok"␣}}'

    subscribe_topic = '/devices/{}/config'.format(nodeID)
    logger.info('Subscribing␣to␣config␣topic:␣{}'.format(
        subscribe_topic))
    _, mid = client.subscribe(subscribe_topic, qos=1)
    response = template.format(nodeID, 'subscribe')
    gateway_state.subscriptions[subscribe_topic] = (nodeID)
    logger.debug('Save␣mid␣{}␣for␣response␣{}'.format(mid,
        response))
    gateway_state.pending_subscribes[mid] = (nodeID, response)

    command_topic = '/devices/{}/commands/#'.format(nodeID)
    logger.info('Subscribing␣to␣command␣topic:␣{}'.format(
        command_topic))
    _, mid = client.subscribe(command_topic, qos=0)
    gateway_state.subscriptions[command_topic] = (nodeID)

    # TODO add suport for error logs
    #logger.info('Subscribing to error topic: {}'.format(
        command_topic))


# handles the configuration updates for the gateway
def handle_config_update(payload):

    if not payload:
        return

    # The config is passed in the payload of the message. In
        this example,
    # the server sends a serialized JSON string.
    logger.info('Received updated config: {}'.format(payload))
    try:
        config = json.loads(payload, encoding='utf-8')
    except ValueError:  # includes simplejson.decoder.
        JSONDecodeError
        logger.error('Decoding JSON in config has failed')
        logger.error('Invalid json: {}'.format(output))
        return

    if not config:
        logger.error('Empty config, Json error: {}'.format(
            output))
        return

    global gateway_state
    gateway_state.update_config(config)


# attaches the device_id to this gateway
# only attached devices can send telemetry to the cloud through
    this gateway
# the device_id needs to be bound to the gateway before
def attach_device(client, device_id):
    attach_topic = '/devices/{}/attach'.format(device_id)
    logger.info('Attach device:{}, {}'.format(device_id,
        attach_topic))
    return client.publish(attach_topic, "", qos=1)


# detaches the device_id from this gateway
def detatch_device(client, device_id):
    detach_topic = '/devices/{}/detach'.format(device_id)
    logger.info('Detatch device:{}, {}'.format(device_id,
        detach_topic))
    return client.publish(detach_topic, "", qos=1)


# handles a system message from a node
def handleSystemMessage(client, nodeID, msg, nodeID_num):
    template = '{{ "device": "{}", "command": "{}", "status" : "
        ok" }}'

    # auth request from a node
    if msg == 'init':

        gateway_state.nodeID_num_for[nodeID] = nodeID_num

        # detach
        _, detach_mid = detatch_device(client, nodeID)
        response = template.format(nodeID, 'detach')
        logger.debug('Save mid {} for response {}'.format(
            detach_mid, response))
        gateway_state.pending_responses[detach_mid] = (
            nodeID, response)

        # attach
        _, attach_mid = attach_device(client, nodeID)
        response = template.format(nodeID, 'attach')
        logger.debug('Save mid {} for response {}'.format(
            attach_mid, response))
        # Pending Subscribes are queued after the mid is acked
            by the server
        gateway_state.pending_attach[attach_mid] = (nodeID,
            response)

        # Needed time for the server to attach device
        # Only then it will accept subscriptions
        # The nodes are attached only when PUBACK is received
            for the nodeID

        # logger.info("Sleeping for 5...")
        # time.sleep(5)

    else:
        logger.warning('undefined system msg: {}'.format(msg))


# handles a telemetry message form a node
def handleReadings(client, nodeID, reading):

    # Adding timestamp, nodeID and derived variables
    reading['timestamp'] = get_formatted_time()
    reading['nodeID'] = nodeID

    # Publishing to IoT for live access
    template = '{{ "device": "{}", "command": "{}", "status" : "
        ok" }}'
    logger.debug('Sending telemetry event for device {}'.format(
        nodeID))

    mqtt_topic = '/devices/{}/events'.format(nodeID)

    logger.info('Publishing message to topic {} with payload
        \'{}\''.format(
        mqtt_topic, json.dumps(reading)))

    if isinstance(reading, str):
        _, event_mid = client.publish(mqtt_topic, reading, qos
            =1)
    else:
        _, event_mid = client.publish(mqtt_topic, json.dumps(
            reading), qos=1)
    response = template.format(nodeID, 'event')
    logger.debug('Save mid {} for response {}'.format(event_mid,
         response))
    gateway_state.pending_responses[event_mid] = (nodeID,
        response)

    # Saving to BigQuery for archival storage
    errors = bqHandler.addToBQ(reading)
    if errors != []:
        logger.error("Error adding to BigQuery")
        logger.error(errors)

    # Adding to logs for local backup
    logger.log(READING_LEVEL, reading)


# sends the message to the nodeID in the mesh
def sendTo(nodeID, message):
    # output format:
    # <numeric_nodeID> <message>

    to_send = '{} {}\n'.format(gateway_state.nodeID_num_for[
        nodeID], message)

    logger.info('Sending {}'.format(to_send))

    # sends the message to the process on which the
        painlessMeshBoost bridge is running which sends the
        payload to the specific node_id
    process.stdin.write(to_send.encode('utf-8'))

# endregion


# region helper functions


def initial_setup(args):
    global gateway_state
    global setupVars
    global logger
    global bqHandler
    global process

    setupLogging(args.log)

    logger.debug("Initating setup")

    gateway_state = GatewayState()

    setupVars.gateway_id = args.gateway_id

    # Subscription topic for configuration updates for the
        gateway
    gateway_state.mqtt_config_topic = '/devices/{}/config'.
        format(
         setupVars.gateway_id)

    gateway_state.mqtt_bridge_hostname = setupVars.
        mqtt_bridge_hostname
    gateway_state.mqtt_bridge_port = setupVars.
        mqtt_bridge_hostname

    setupVars.path_to_bqcreds = args.json_creds

    bqHandler = BigQueryHandler()

    client = get_client(
        setupVars.project_id, setupVars.cloud_region, setupVars.
            registry_id,
        setupVars.gateway_id, args.private_key_file, setupVars.
            algorithm,
        args.ca_certs, setupVars.mqtt_bridge_hostname, setupVars
            .mqtt_bridge_port,
        setupVars.jwt_expires_minutes)

    return client


# parses command line arguments
def parse_command_line_args():
    kdd = keys_dir
    if is_windows():
        kdd = kdd + '\\'
    else:
        kdd = kdd+'/'

    parser = argparse.ArgumentParser(description=(
        'Example␣Google␣Cloud␣IoT␣Core␣MQTT␣device␣connection␣
            code.'))
    parser.add_argument(
        '--gateway_id', required=True, help='Gateway␣Name/ID')
    parser.add_argument(
        '--private_key_file', default='{}rsa_private.pem'.format
            (kdd),
        help='Path␣to␣private␣key␣file.')
    parser.add_argument(
        '--ca_certs', default='{}roots.pem'.format(kdd),
        help=('CA␣root␣from␣https://pki.google.com/roots.pem'))
    parser.add_argument(
        '--json_creds', default='{}bq-manager.json'.format(kdd),
        help=('Path␣to␣json␣creds␣for␣a␣Service␣Account␣with␣r/w
            ␣access␣to␣BigQuery'))
    parser.add_argument(
        '--log',
        choices=('reading', 'all', 'debug', 'info',
                 'warning', 'error', 'critical'),
        default='error',
        help='Console output logging.')
    parser.add_argument(
        '--interface',
        default='wlan0',
        help='Network interface through which the gateway is 
            connected to the mesh')
    return parser.parse_args()


# helper function to get the proper formatted time with timezone
     to store as timestamp in the BigQuery database
def get_formatted_time(timezone='Asia/Kolkata'):
    tz = datetime.datetime.now(pytz.timezone(timezone)).strftime
        ("%z")
    tz = tz[:3]+':'+tz[3:]
    timestamp = datetime.datetime.now(pytz.timezone(
        'Asia/Kolkata')).strftime("%Y-%m-%d %H:%M:%S.%f")+tz
    return timestamp


# helper function to set up logging
def setupLogging(cLevel):
    print("setting log level to {}".format(cLevel))
    custom = False
    if cLevel == 'debug':
        cLevel = logging.DEBUG
    elif cLevel == 'info':
        cLevel = logging.INFO
    elif cLevel == 'warning':
        cLevel = logging.WARNING
    elif cLevel == 'error':
        cLevel = logging.ERROR
    elif cLevel == 'critical':
        cLevel = logging.CRITICAL
    elif cLevel == 'all':
        cLevel = READING_LEVEL
    elif cLevel == 'reading':
        cLevel = logging.ERROR
        custom = True

    class LogFilter(object):
        def __init__(self, level, level2=None):
            self.__level = level
            self.__level2 = level2

        def filter(self, logRecord):
            # handler1.addFilter(MyFilter(logging.INFO))
            # handler2.addFilter(MyFilter(logging.ERROR))
            if not self.__level2:
                return logRecord.levelno <= self.__level
            else:
                return logRecord.levelno <= self.__level or
                    logRecord.levelno >= self.__level2
```

```
global logger
# create logger with 'gatway-root'
logging.addLevelName(READING_LEVEL, 'Reading')
logger = logging.getLogger('gatway-root')
logger.setLevel(READING_LEVEL)

path = os.getcwd()
path = path + 'logs'
try:
    os.mkdir('logs')
except FileExistsError:
    print("Logs␣directory␣already␣exists")
except OSError:
    sys.exit("Creation␣of␣the␣directory␣%s␣failed" % path)

# create file handler which logs even debug messages
fh = logging.FileHandler('./logs/full-logs.log')
fh.setLevel(logging.DEBUG)

# create file handler which logs just errors
eh = logging.FileHandler('./logs/error-log.log')
eh.setLevel(logging.ERROR)

# create file handler which logs just readings
rh = logging.FileHandler('./logs/reading.log')
rh.setLevel(READING_LEVEL)
rh.addFilter(LogFilter(READING_LEVEL))

# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(cLevel)

if custom:
    ch2 = logging.StreamHandler()
    ch2.setLevel(READING_LEVEL)
    ch2.addFilter(LogFilter(READING_LEVEL))
    ch2Format = logging.Formatter('%(asctime)s␣-␣%(message)s
        ')
    ch2.setFormatter(ch2Format)
    logger.addHandler(ch2)

# create formatter and add it to the handlers
formatter = logging.Formatter(
    '%(asctime)s␣-␣%(funcName)s␣-␣%(levelname)s␣-␣%(message)
        s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
eh.setFormatter(formatter)
readingFormat = logging.Formatter('%(asctime)s␣-␣Packet:␣[%(
    message)s]')
rh.setFormatter(readingFormat)

# add the handlers to the logger
logger.addHandler(fh)
```

```
    logger.addHandler(ch)
    logger.addHandler(rh)
    logger.addHandler(eh)


# get wifi ip address
def get_ip_address(args):
    win = is_windows()
    wlan_interface = args.interface

    if win:
        gw = os.popen(
            ”netsh␣interface␣ip␣show␣config␣name=\”WiFi\”␣|␣
                findstr␣\”Default␣Gateway:\””).read().split()
    else:
        gw = os.popen(
            ”ip␣−4␣address␣show␣dev␣{}␣|␣grep␣−e␣\”inet\””.
                format(wlan_interface)).read().split()

    while(not gw):
        print(”Waiting␣for␣wifi...”)
        sleep(1)
        if win:
            gw = os.popen(
                ”netsh␣interface␣ip␣show␣config␣name=\”WiFi\”␣|␣
                    findstr␣\”Default␣Gateway:\””).read().split()
        else:
            gw = os.popen(
                ”ip␣−4␣address␣show␣dev␣{}␣|␣grep␣−e␣\”inet\””.
                    format(wlan_interface)).read().split()

    if win:
        ipaddress = gw[2]
    else:
        ipa = gw[1].split('/')[0].split('.')
        # get gateway ip
        ipaddress = ipa[0]+'.'+ipa[1]+'.'+ipa[2]+'.1'

    return ipaddress


# checks whether the operating system is Windows or not
def is_windows():
    if ”Windows” in platform.platform():
        return True
    return False

# endregion


def main():
    global gateway_state
    global logger
    global process
```

```
args = parse_command_line_args()

client = initial_setup(args)

ipaddress = get_ip_address(args)

# runs the painlessMeshBoost bridge as a subprocess with
    which we can communicate
# and receive messages from the nodes in the mesh
command = 'painlessMeshBoost -n 696969 --client '+ipaddress
if is_windows():
    process = subprocess.Popen(
        shlex.split(command),
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        stdin=subprocess.PIPE,
        shell=True,
        bufsize=0)
else:
    process = subprocess.Popen(
        shlex.split(command),
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        stdin=subprocess.PIPE,
        bufsize=0)

while True:
    client.loop()
    if gateway_state.connected is False:
        logger.info('connect status {}'.format(gateway_state
            .connected))
        time.sleep(1)
        continue

    # command output
    output = process.stdout.readline()

    if output == '' and process.poll() is not None:
        continue

    output = str(output.strip(), encoding='utf-8')
    if output == "" or 'setLogLevel' in output:
        continue

    try:
        logline = json.loads(output)
    except ValueError:  # includes simplejson.decoder.
        JSONDecodeError
        logger.warning('Decoding JSON has failed')
        logger.error('invalid json: {}'.format(output))
        continue

    if not logline:
        logger.error('invalid json syntax: {}'.format(output
            ))
        continue

    event = logline['event']

    if event == 'change' or event == 'connect' or event == '
        offset':
        # debug log
        logger.debug('Mesh Log: [{}]'.format(logline))

    elif event == 'receive':
        # direct message
        TYPE_DEBUG = "debug"
        TYPE_ERROR = "error"
        TYPE_SYSTEM = "system"
        TYPE_DATA = "data"

        datastr = logline['msg']

        # ignore broadcast
        if datastr == 'gatewayConnected':
            continue

        logger.debug('Received message: [{}]'.format(logline
            ))

        try:
            data = json.loads(datastr)
        except ValueError:  # includes simplejson.decoder.
            JSONDecodeError
            logger.warning('Decoding data JSON has failed')
            logger.error('invalid json: {}'.format(datastr))
            continue

        if not data:
            logger.error('invalid data received: {}'.format(
                output))
            continue

        ty = data['type']
        nodeID = data['nodeID']
        msg = data['data']
        nodeID_num = logline['nodeId']

        logger.info('Received Data: {} from Node: {}'.format
            (data, nodeID))

        if ty == TYPE_SYSTEM:
            handleSystemMessage(client, nodeID, msg,
                nodeID_num)
        elif ty == TYPE_DATA:
            handleReadings(client, nodeID, msg)
        else:
                logger.warning("Unknown message type '{}'".
                    format(ty))
                logger.war
        else:
            logger.warning(
                "Unknown event: {}, details: {}".format(event,
                    logline))

if __name__ == '__main__':
    main()
```

Bibliography

[1] Gourinath Banda, Krishna Chaitanya, and Harsh Mohan. “An IoT Protocol and Framework for OEMs to Make IoT-Enabled Devices forward Compatible”. In: *11th International Conference on Signal-Image Technology & Internet-Based Systems, SITIS 2015, Bangkok, Thailand, November 23-27, 2015*. Ed. by Kokou Yétongnon and Albert Dipanda. IEEE Computer Society, 2015, pp. 824–832. ISBN: 978-1-4673-9721-6. DOI: `10.1109/SITIS.2015.106`. URL: `https://doi.org/10.1109/SITIS.2015.106`.

[2] S. R. Prathibha, Anupama Hongal, and M. P. Jyothi. “IOT Based Monitoring System in Smart Agriculture”. In: *2017 International Conference on Recent Advances in Electronics and Communication Technology (ICRAECT)*. IEEE, Mar. 2017. DOI: `10.1109/icraect.2017.52`.

[3] Tomo Popović et al. “Architecting an IoT-enabled platform for precision agriculture and ecological monitoring: A case study”. In: *Computers and Electronics in Agriculture* 140 (Aug. 2017), pp. 255–265. DOI: `10.1016/j.compag.2017.06.008`.

[4] I. Ganchev, Zhanlin Ji, and M. O’Droma. “A Generic IoT Architecture for Smart Cities”. In: *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communities Technologies (ISSC 2014/CIICT 2014)*. Institution of Engineering and Technology, 2014. DOI: `10.1049/cp.2014.0684`.

[5] S. Ivanov, K. Bhargava, and W. Donnelly. “Precision Farming: Sensor Analytics”. In: *IEEE Intelligent Systems* 30.4 (July 2015), pp. 76–80. ISSN: 1941-1294. DOI: `10.1109/MIS.2015.67`.

[6] N. Ahmed, D. De, and I. Hussain. “Internet of Things (IoT) for Smart Precision Agriculture and Farming in Rural Areas”. In: *IEEE Internet of Things Journal* 5.6 (Dec. 2018), pp. 4890–4899. ISSN: 2372-2541. DOI: `10.1109/JIOT.2018.2879579`.

[7] A. Khattab, A. Abdelgawad, and K. Yelmarthi. “Design and implementation of a cloud-based IoT scheme for precision agriculture”. In:

2016 28th International Conference on Microelectronics (ICM). Dec. 2016, pp. 201–204. DOI: 10.1109/ICM.2016.7847850.

[8] Google. *Using gateways (accessed on 01/12/2019)*. URL: https://cloud.google.com/iot/docs/how-tos/gateways/.

[9] Open Automation Software. *IoT Gateways – Powering the Industrial Internet of Things (accessed on 01/12/2019)*. URL: https://openautomationsoftware.com/blog/what-is-an-iot-gateway/.

[10] Google. *Introducing Google Cloud IoT Core: for securely connecting and managing IoT devices at scale (accessed on 01/12/2019)*. 2017. URL: https://cloud.google.com/blog/products/gcp/introducing-google-cloud-iot-core-for-securely-connecting-and-managing-iot-devices-at-scale.

[11] Peerapak Lerdsuwan and Phond Phunchongharn. "An Energy-Efficient Transmission Framework for IoT Monitoring Systems in Precision Agriculture". In: *Information Science and Applications 2017*. Springer Singapore, 2017, pp. 714–721. DOI: 10.1007/978-981-10-4154-9_82.

[12] PlatformIO. *A new generation ecosystem for embedded development (accessed on 01/12/2019)*. URL: https://platformio.org/.

[13] BlackEdder. *painlessMesh - setup a mesh with ESP8266 and ESP32 devices (accessed on 01/12/2019)*. URL: https://gitlab.com/painlessMesh/painlessMesh.

[14] BlackEdder. *painlessMeshBoost - runs painlessMesh node on any platform supported by boost (accessed on 01/12/2019)*. URL: https://gitlab.com/painlessMesh/painlessmeshboost.