Final Report
CS3242 Micro-controllers and Applications

Environment Monitoring IoT Device with Common Alert Protocol

K. D. Sunera Avinash Chandrasiri
170081L

October 30, 2020

Contents

1 Introduction

1.1 Scope

This project is a device developed using the **NodeMCU 32S** micro-controller, which monitors environment parameters such as **temperature, humidity, barometric pressure, and ambient light level**. The used micro-controller will operate with low-power and via a wi-fi connection. Even though the wi-fi connection network availability may be unreliable, firmware for the microcontroller is designed in a way so that network unreliability would not entirely hinder the operation of the sensor.

The device will connect with a remote server via an HTTP request. Here the microcontroller will use **CAP(Common Alert Protocol)** to deliver the relevant parameters. This "server sync" will happen once every 15 minutes. In these situations, the device will send the mean and the standard deviation of the samples the device collected within that 15 minutes. If the network connection fails for some reason, the device would still cache the requests(up to a certain extent) and resend/replay those messages once the network reconnects.

The backend implemented for the project is a simple server based off of python. It uses **flask** framework and sqlite database. It simply accepts alert notifications sent by the device and displays them in a tabular manner.

1.2 Features

Following are some of the features available in the device. Node that following list also includes the features that were expected from the project as well as some special features.

- Measuring temperature, humidity, barometric pressure, and ambient light level.
- Syncing with the server every 15 minutes.
- Use Common Alert Protocol 1.2 to deliver the messages.
- Replay messages that failed to deliver to the server due to network issues.
- In case of replaying past messages, timestamp is also sent after retreiveing it via NTP, so the server can correctly order the messages.
- Create a unique session identifier at the initialization of the device.
- Support multi-device connections. Because of the above unique identifier, we can connect multiple devices to the server.

2 Design

2.1 High Level Design

Following the high level system diagram is shown. The system will mainly consist of two major parts; embedded device and the remote server. The device will use a Wi-Fi connection to connect to the remote server.

Figure 1: High Level Deployment Diagram

The high-level diagram of the embedded device is shown below. Here, the sensors required are connected; BMP180 (for pressure sensing), DHT11 (for humidity and temperature sensing) and LDR(for light level sensing).

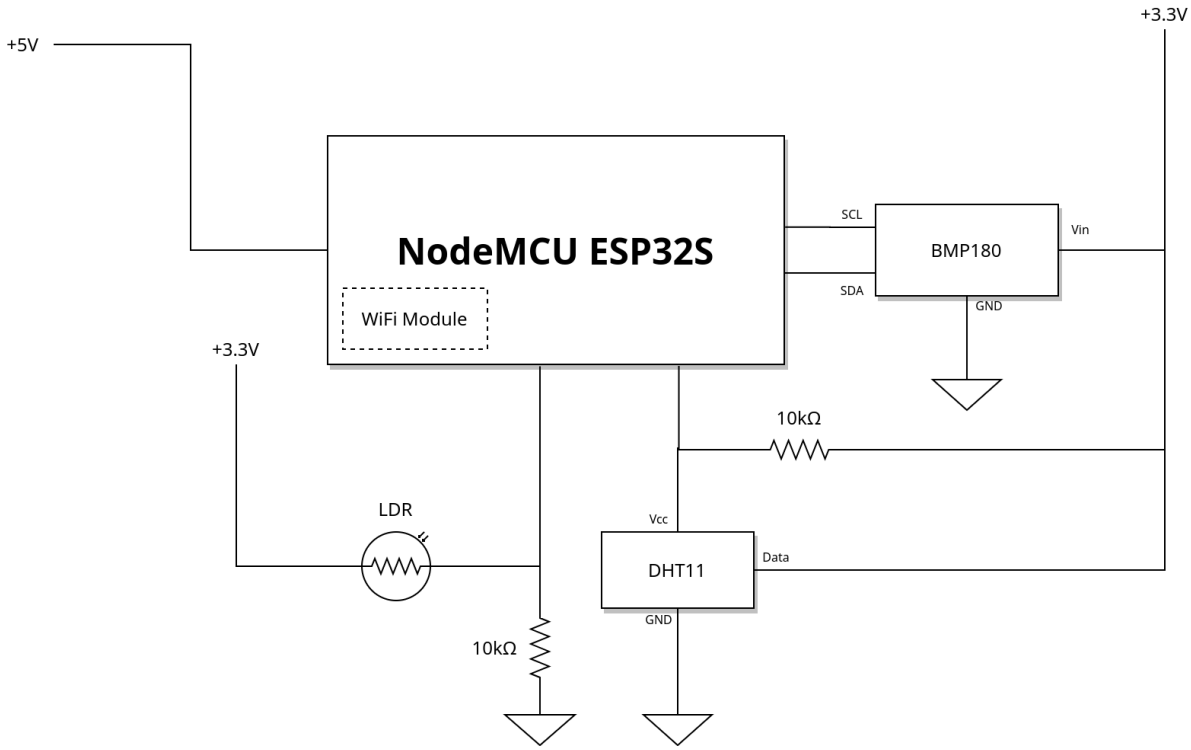

Figure 2: High Level Device Diagram

The device will run on 5V input voltage while the input/output high voltage of the NodeMCU will be 3.3V.

2.2 Components and Cost Breakdown

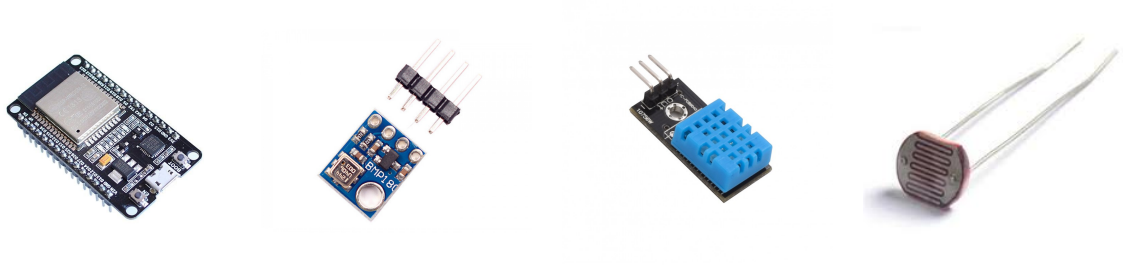

Figure 3: From left to right - NodeMCU, BMP180, DHT11, LDR

The Price Breakdown for the product components are given below. The chosen microcontroller board was **NodeMCU ESP-32S IoT Dev Board** mainly because of its inbuilt Wi-Fi capabilities and its specialization on such tasks, especially on IoT field. Apart form that, for the sensors, following sensors were chosen.

DHT11 Sensor	Temperature and Relative Humidity Sensor Module. Temperature range is 0°C to 50°C (±1°C) and Humidity Range is 20% to 90% (±1%) Operating voltage is 3.5V to 5.5V.
BMP180 Sensor	Barometric Pressure sensor module. Range is 300hPa to 1100 hPa with 0.02 hPa accuracy. Operating voltage is 1.8-6 V.
LDR	Light Dependent Resistor for measuring ambient light level.

Cost breakdown for the device excluding connectors, vero-boards, etc... is given below.

Component Name	**Subtotal (LKR)**
NodeMCU ESP-32S WiFi Bluetooth Dual Mode IoT Dev Board	1050.00
BMP180 Barometric Pressure, Temperature and Altitude Sensor	185.00
DHT11 Temperature and Relative Humidity Sensor Module	250.00
LDR 4mm	10.00
Total Amount (LKR)	**1495.00**

The cost could have been further reduced by using a **NodeMCU ESP-8266** board. This costs about half of the NodeMCU ESP-32S price. However, ESP-32S was cited as being more reliable than ESP-8266. So, ESP-32S was used.

2.3 Schematic Diagram

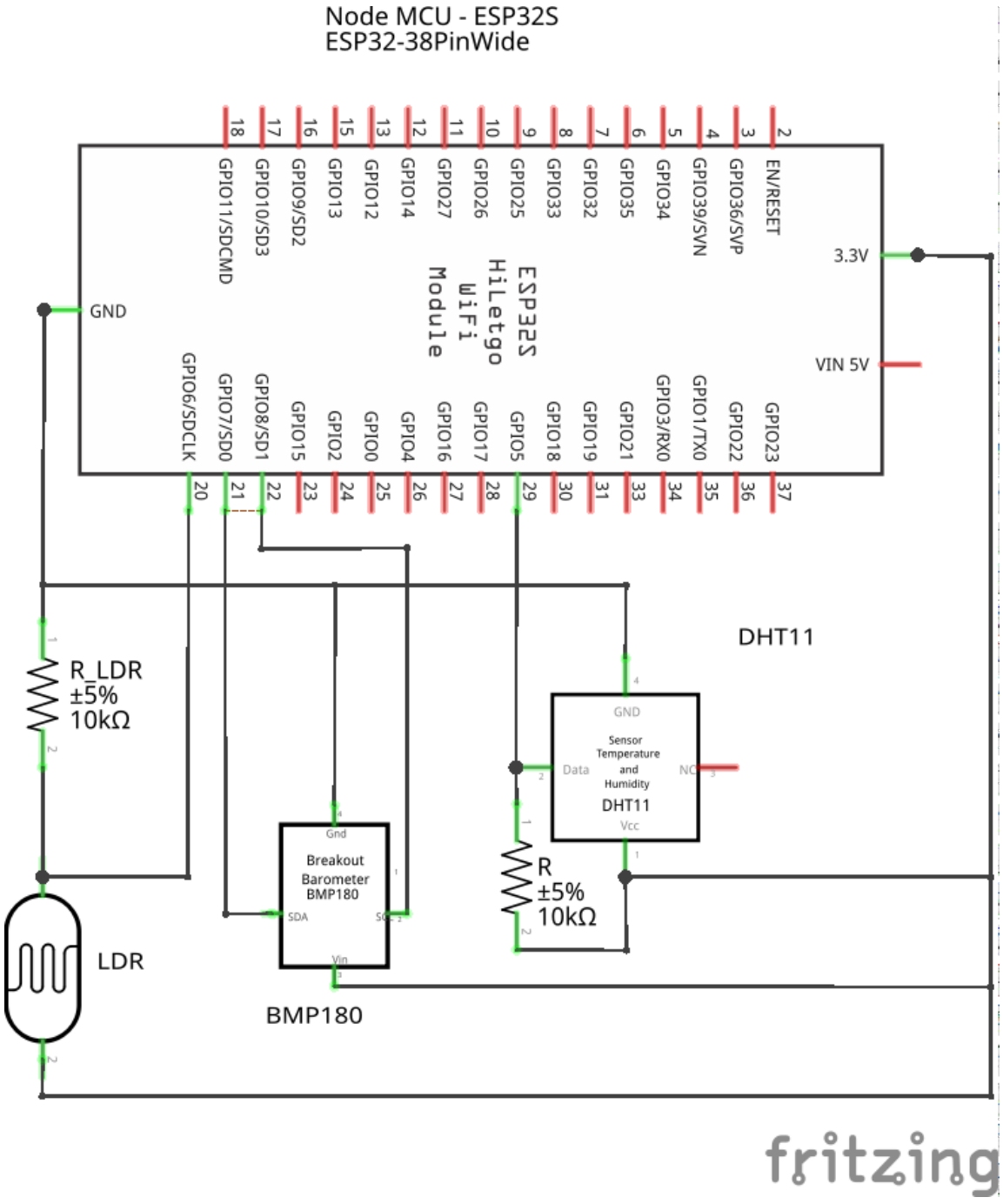

Figure 4: Schematic Diagram of the device

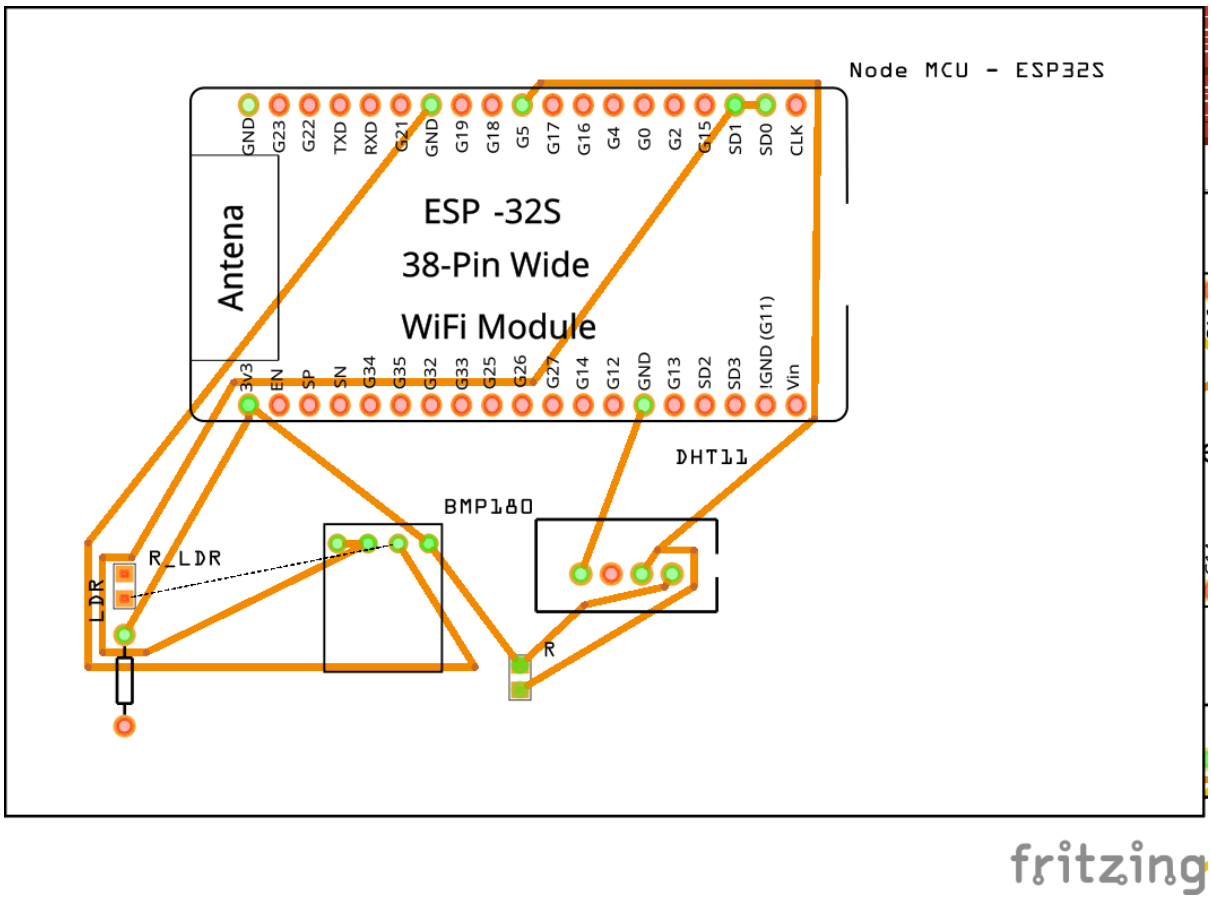

Figure 5: Schematic PCB view of the prototype

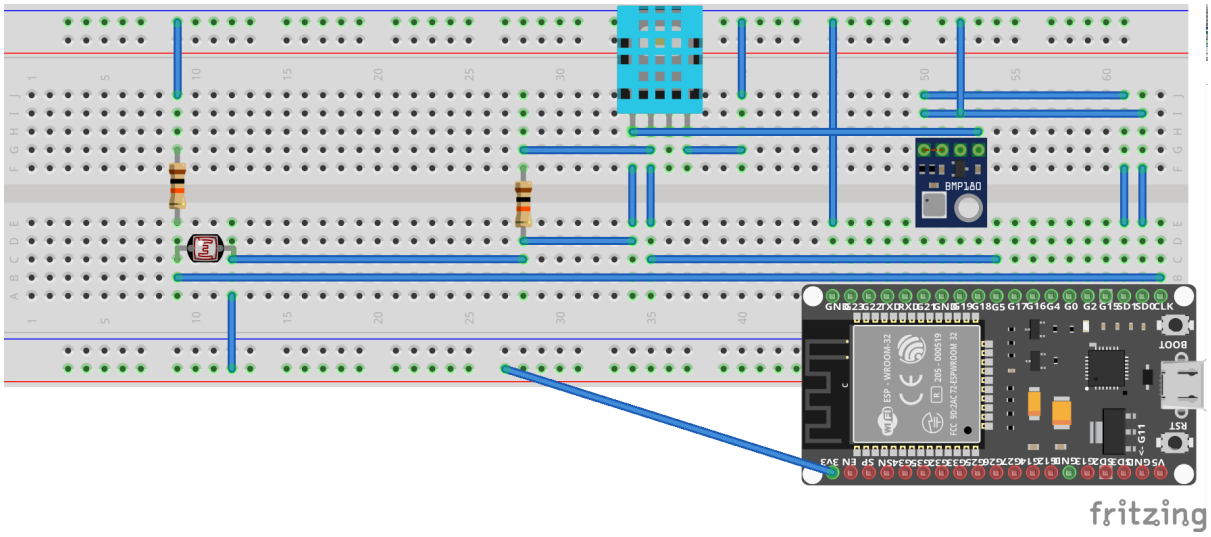

Figure 6: Breadboard view of the prototype

3 Implementation

3.1 Fault Recovery Implementation

3.2 Algorithm

Algorithm 1: Pseudo code of the algorithm implemented

```
queue ⟵ LifoQueue()
tempSamples ⟵ array()
humiditySamples ⟵ array()
pressureSamples ⟵ array()
lightSamples ⟵ array()
samplesCollected ⟵ 0

Function sendSamples() is
    // find mean & std
    tempMean ⟵ mean(tempSamples)
    // ...
    lightStd ⟵ std(tempSlightSamplesamples)
    // Add message to end of queue
    cap ⟵ createCapMessage(tempMean, ..., lightStd)
    if queue.isFull() then
        queue.pop()
    end
    queue.push(cap)
    // send all current messages
    currentQueueSize ⟵ queue.size()
    for i ← 0 to currentQueueSize do
        capEntry ⟵ queue.pop()
        response ⟵ wifi.sendPostRequest(capEntry)
        // Re-add failed requests
        if response = ERR ∨ response.status ≠ 200 then
            queue.push(capEntry)
        end
    end
    // reset variables
    samplesCollected ← 0
    tempSamples.clear()
    // ...
    lightSamples.clear()
end

Function collectSamples() is
    temp ← readDHT11()
    humidity ← readDHT11()
    pressure ← readBMP180()
    light ← readLDR()
    tempSamples.add(temp)
    // ...
    lightSamples.add(light)
    samplesCollected++
end

begin
    while True do
        if timeSinceLastRead() ≥ Δ then
            if samplesCollected == N then sendSamples()
            collectSamples()
        end
    end
end
```

4 Annexure

4.1 Server Source Code

REMOTE_SERVER/app.py

```
from flask import Flask
from flask import request
from capparselib.parsers import CAPParser
from flask import render_template

from db import database, entry

app = Flask(__name__)
database.init_app(app)


@app.route('/', methods=["POST"])
def get_update():
    alerts = CAPParser(request.data.decode('utf-8')).as_dict()
    alert = alerts[0]
    params = {}
    for param in alert['cap_info'][0]['cap_parameter']:
        name = param['valueName']
        value = param['value']
        params[name] = value
    entry.save_entry(alert, params)
    return "200 - ACK"


@app.route('/', methods=["GET"])
def view():
    entries = entry.get_all_entries()
    return render_template('index.html', entries=entries)


if __name__ == '__main__':
    app.run()
```

REMOTE_SERVER/db/entry.py

```
from db.database import get_db
from datetime import datetime


def get_all_entries():
    c = get_db().cursor()
    c.execute("""
    select identifier, CAST(sent as TEXT) as sent,
               info_parameter_temp_mean, info_parameter_temp_std,
               info_parameter_humidity_mean, info_parameter_humidity_std,
               info_parameter_pressure_mean, info_parameter_pressure_std,
               info_parameter_light_mean, info_parameter_light_std
    from entry order by sent;
    """)
    return c.fetchall()


def save_entry(cap, params):
    conn = get_db()
    c = conn.cursor()
    c.execute("""
        INSERT INTO entry (identifier, sender, sent, status, msgType, scope,
                           info_category, info_event, info_responseType,
                           info_urgency, info_severity, info_certainty, info_senderName,
                           info_parameter_temp_mean,  info_parameter_temp_std,
                           info_parameter_humidity_mean, info_parameter_humidity_std,
                           info_parameter_pressure_mean, info_parameter_pressure_std,
                           info_parameter_light_mean, info_parameter_light_std)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
    """,
              (
                  str(cap['cap_id']),
                  str(cap['cap_sender']),
                  datetime.fromisoformat(str(cap['cap_sent'])),
                  str(cap['cap_status']),
                  str(cap['cap_message_type']),
                  str(cap['cap_scope']),
                  str(cap['cap_info'][0]['cap_category']),
                  str(cap['cap_info'][0]['cap_event']),
                  str(cap['cap_info'][0]['cap_response_type']),
                  str(cap['cap_info'][0]['cap_urgency']),
                  str(cap['cap_info'][0]['cap_severity']),
                  str(cap['cap_info'][0]['cap_certainty']),
                  str(cap['cap_info'][0]['cap_sender_name']),
                  float(params['TEMP_MEAN']),
                  float(params['TEMP_STD']),
                  float(params['HUMIDITY_MEAN']),
                  float(params['HUMIDITY_STD']),
                  float(params['PRESSURE_MEAN']),
                  float(params['PRESSURE_STD']),
                  float(params['LIGHT_MEAN']),
                  float(params['LIGHT_STD']),
              ))
    conn.commit()
```

REMOTE_SERVER/db/database.py

```
import sqlite3

import click
from flask import current_app, g
from flask.cli import with_appcontext


def init_db():
    db = get_db()
    with current_app.open_resource('schema.sql') as f:
        db.executescript(f.read().decode('utf8'))


@click.command('init-db')
@with_appcontext
def init_db_command():
    """Clear the existing data and create new tables."""
    init_db()
    click.echo('Initialized the database.')


def init_app(app):
    app.teardown_appcontext(close_db)
    app.cli.add_command(init_db_command)


def get_db():
    if 'db' not in g:
        g.db = sqlite3.connect(
            'db/database.db',
            detect_types=sqlite3.PARSE_DECLTYPES
        )
        g.db.row_factory = sqlite3.Row

    return g.db


def close_db(e=None):
    db = g.pop('db', None)
    if db is not None:
        db.close()
```

4.2 Device Source Code

FIRMWARE/src/main.cpp

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

#include "cap/cap.h"
#include "sensor/bmp180.h"
#include "sensor/dht.h"
#include "sensor/sensor.h"
#include "utils.h"
#include "wifi/controller.h"
#include "wifi/ntp.h"

#define CAP_SIZE 3000
#define N_SAMPLES 5
#define CACHE_SIZE 10
#define REMOTE_SERVER "http://7e86ccb1d82f.ngrok.io/"

#define SYNC_PERIOD (30 * 1000)
#define SAMPLE_PERIOD (SYNC_PERIOD / (N_SAMPLES))

float tempSamples[N_SAMPLES];
float humiditySamples[N_SAMPLES];
float pressureSamples[N_SAMPLES];
float lightSamples[N_SAMPLES];
int currentSample = 0;
unsigned long lastSampleMillis;
Queue capQueue(CAP_SIZE, CACHE_SIZE);

void clearSample(float* buffer) {
  for (int i = 0; i < N_SAMPLES; i++) {
    buffer[i] = NULL;
  }
}

void clearSamples() {
  clearSample(tempSamples);
  clearSample(humiditySamples);
  clearSample(pressureSamples);
  clearSample(lightSamples);
}

statistic toStatistic(float* buffer) {
  float sensorMean = mean(buffer, N_SAMPLES);
  float sensorStd = standardDeviation(buffer, N_SAMPLES, sensorMean);
  return {sensorMean, sensorStd};
}

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(115200);
  initializeConnection();
  initializeNtp();
  setupIdentifier();
  initializeDHT();
  initializeBMP();
  clearSamples();
```

```
}

void loop() {
  if (millis() - lastSampleMillis < SAMPLE_PERIOD) {
    return;
  }

  lastSampleMillis = millis();
  if (currentSample == N_SAMPLES) {
    Serial.println("[MAIN] Starting server sync...");
    char cap[CAP_SIZE];
    statistic temp = toStatistic(tempSamples);
    statistic humidity = toStatistic(humiditySamples);
    statistic pressure = toStatistic(pressureSamples);
    statistic light = toStatistic(lightSamples);
    generateCap(cap, CAP_SIZE, temp, humidity, pressure, light);
    if (capQueue.isFull()) {
      Serial.println("[CACHE] CAP cache full, discarding oldest entry.");
      capQueue.drop();
    }
    capQueue.push(cap);
    char serverAddress[] = REMOTE_SERVER;
    processCapQueue(&capQueue, CAP_SIZE, serverAddress);
    currentSample = 0;
    clearSamples();
  } else {
    Serial.println("[MAIN] Starting sensor sync...");
    float temp = sampleTemperature();
    float humidity = sampleHumidity();
    float pressure = samplePressure();
    float light = sampleLight();
    tempSamples[currentSample] = temp;
    humiditySamples[currentSample] = humidity;
    pressureSamples[currentSample] = pressure;
    lightSamples[currentSample] = light;
    currentSample++;
  }
}
```

FIRMWARE/src/utils.h

```
#ifndef UTILS_H
#define UTILS_H

float mean(float* buffer, int size);
float standardDeviation(float* buffer, int size, float mean);

#endif
```

FIRMWARE/src/utils.cpp

```
#include <math.h>

float mean(float* buffer, int size) {
  float sum = 0.0;
  int count = 0;
  for (int i = 0; i < size; i++) {
    if (buffer[i] >= 0) {
      count++;
      sum += buffer[i];
    }
  }
  if (count == -1) return -1;
  return sum / count;
}

float standardDeviation(float* buffer, int size, float mean) {
  float std = 0.0;
  int count = 0;
  for (int i = 0; i < size; i++) {
    if (buffer[i] >= 0) {
      count++;
      std += pow(buffer[i] - mean, 2);
    }
  }
  if (count == -1) return -1;
  return pow(std / count, 0.5);
}
```

FIRMWARE/src/cap/cap.h

```
#ifndef CAP_H
#define CAP_H

#include "statistic.h"

void generateCap(char* buffer, int buffer_size, statistic temp,
                 statistic humidity, statistic pressure, statistic light);
void setupIdentifier();

#endif
```

FIRMWARE/src/cap/statistic.h

```
#ifndef STATISTIC_H
#define STATISTIC_H

struct statistic {
  float mean;
  float std;
};

#endif
```

FIRMWARE/src/cap/cap.cpp

```
#include <Arduino.h>
#include <esp_system.h>
#include <stdio.h>

#include "../wifi/ntp.h"
#include "statistic.h"

#define IDENTIFIER_LENGTH 32

const char* CAP_TEMPLATE =
    "<?xml version = \"1.0\" encoding = \"UTF-8\"?>"
    "<alert xmlns = \"urn:oasis:names:tc:emergency:cap:1.2\">"
    "   <identifier>%s-%d</identifier>"
    "   <sender>MCU_DEVICE</sender>"
    "   <sent>%s</sent>"
    "   <status>Actual</status>"
    "   <msgType>Alert</msgType>"
    "   <scope>Private</scope>"
    "   <info>"
    "       <category>Env</category>"
    "       <event>SENSOR_DATA</event>"
    "       <responseType>None</responseType> "
    "       <urgency>Unknown</urgency>"
    "       <severity>Minor</severity>"
    "       <certainty>Observed</certainty>"
    "       <senderName>mcu_session_%s</senderName>"
    "       <parameter>"
    "           <valueName>TEMP_MEAN</valueName>"
    "           <value>%f</value>"
    "       </parameter>"
    "       <parameter>"
```

```
    "            <valueName>TEMP_STD</valueName>"
    "            <value>%f</value>"
    "        </parameter>"
    "        <parameter>"
    "            <valueName>HUMIDITY_MEAN</valueName>"
    "            <value>%f</value>"
    "        </parameter>"
    "        <parameter>"
    "            <valueName>HUMIDITY_STD</valueName>"
    "            <value>%f</value>"
    "        </parameter>"
    "        <parameter>"
    "            <valueName>PRESSURE_MEAN</valueName>"
    "            <value>%f</value>"
    "        </parameter>"
    "        <parameter>"
    "            <valueName>PRESSURE_STD</valueName>"
    "            <value>%f</value>"
    "        </parameter>"
    "        <parameter>"
    "            <valueName>LIGHT_MEAN</valueName>"
    "            <value>%f</value>"
    "        </parameter>"
    "        <parameter>"
    "            <valueName>LIGHT_STD</valueName>"
    "            <value>%f</value>"
    "        </parameter>"
    "    </info>"
    "</alert>";

int currentIdentifier = 0;
char randomIdentifier[IDENTIFIER_LENGTH + 1];

void setupIdentifier() {
  randomSeed(analogRead(0));
  for (int i = 0; i < IDENTIFIER_LENGTH; i++) {
    byte randomValue = random(0, 26);
    randomIdentifier[i] = randomValue + 'a';
  }
  Serial.printf("[CAP] Identifier: %s\n", randomIdentifier);
}

void generateCap(char* buffer, int buffer_size, statistic temp,
                 statistic humidity, statistic pressure, statistic light) {
  updateTime();

  Serial.println("[CAP] Compiling protocol XML.");
  snprintf(buffer, buffer_size, CAP_TEMPLATE, randomIdentifier,
           currentIdentifier++, formattedDate, randomIdentifier, temp.mean,
           temp.std, humidity.mean, humidity.std, pressure.mean, pressure.std,
           light.mean, light.std);
}
```

FIRMWARE/src/sensor/bmp180.h

```
#ifndef BMP180_H
#define BMP180_H

void initializeBMP();
float readPressure();

#endif
```

FIRMWARE/src/sensor/dht.h

```
#ifndef DHT_H
#define DHT_H

void initializeDHT();
float readHumidity();
float readTemperature();

#endif
```

FIRMWARE/src/sensor/sensor.h

```
#ifndef SENSOR_H
#define SENSOR_H

float sampleTemperature();
float sampleHumidity();
float samplePressure();
float sampleLight();

#endif
```

FIRMWARE/src/sensor/bmp180.cpp

```
#include <Adafruit_BMP085.h>
#include <Wire.h>

Adafruit_BMP085 bmp;

void initializeBMP() {
  if (!bmp.begin()) {
    Serial.println("[BMP] Could not find a valid BMP180 sensor.");
  }
}

float readPressure() {
  if (!bmp.begin()) {
    Serial.println("[DHT] Failed to read pressure from BMP sensor.");
    return -1;
  }

  float pressure = bmp.readPressure();
  Serial.printf("[BMP] Pressure read: %f\n", pressure);
  return pressure;
}
```

FIRMWARE/src/sensor/dht.cpp

```cpp
#include <Adafruit_Sensor.h>
#include <Arduino.h>
#include <DHT.h>

#define DHTPIN 4
#define DHTTYPE DHT11

DHT dht(DHTPIN, DHTTYPE);

void initializeDHT() { dht.begin(); }

float readHumidity() {
  float humidity = dht.readHumidity();
  if (isnan(humidity)) {
    Serial.println("[DHT] Failed to read humidity from DHT sensor.");
    return -1;
  }

  Serial.printf("[DHT] Humidity read: %f\n", humidity);
  return humidity;
}

float readTemperature() {
  float temperature = dht.readTemperature();
  if (isnan(temperature)) {
    Serial.println("[DHT] Failed to read temperature from DHT sensor.");
    return -1;
  }

  Serial.printf("[DHT] Temperature read: %f\n", temperature);
  return temperature;
}
```

FIRMWARE/src/sensor/sensor.cpp

```cpp
#include <Arduino.h>

#include "bmp180.h"
#include "dht.h"

float sampleTemperature() { return readTemperature(); }

float sampleHumidity() { return readHumidity(); }

float samplePressure() { return readPressure(); }

float sampleLight() { return 400 + (random(10) * 2 - 5); }
```

FIRMWARE/src/wifi/config.h

```h
#ifndef WIFI_CONFIG_H
#define WIFI_CONFIG_H

extern const char* SSID;
extern const char* PASSWORD;

#endif
```

FIRMWARE/src/wifi/controller.h

```h
#ifndef CONFIG_H
#define CONFIG_H

#include <cppQueue.h>

#include "config.h"

void initializeConnection();
bool sendPostRequest(char* message, char* endpoint);
void processCapQueue(Queue* capCache, int capSize, char* endpoint);

#endif
```

FIRMWARE/src/wifi/ntp.h

```h
#ifndef NTP_H
#define NTP_H

#include <WString.h>

extern String formattedDate;
extern String dayStamp;
extern String timeStamp;

void initializeNtp();
void updateTime();

#endif
```

FIRMWARE/src/wifi/controller.cpp

```cpp
#include "controller.h"

#include <HTTPClient.h>
#include <WiFi.h>
#include <cppQueue.h>

#include "../cap/cap.h"

/**
 * Initializing wifi connection.
 * Configuring wifi module and connecting to the network.
 * */
void initializeConnection() {
  WiFi.begin(SSID, PASSWORD);
  Serial.print("[WIFI] Trying to connect...");
  delay(1000);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  const char* ip = WiFi.localIP().toString().c_str();
  Serial.printf("\n[WIFI] Connected to network with IP: %s\n", ip);
}

/**
 * Sending the message body given to the endpoint
 * as a POST request.
 * Will return false if attempt failed.
 * */
bool sendPostRequest(char* message, char* endpoint) {
  if (WiFi.status() != WL_CONNECTED) {
    Serial.println("[WIFI] Not connected to a network.");
    return false;
  }

  // Send POST request
  Serial.println("[WIFI] Sending POST request to the server...");
  HTTPClient http;
  http.begin(endpoint);
  int responseCode = http.POST(message);
  String response = http.getString();
  http.end();

  // Connection failed error codes
  if (responseCode < 0) {
    String errorMessage = http.errorToString(responseCode).c_str();
    Serial.printf("[WIFI] Request failed: %s\n", errorMessage.c_str());
    return false;
  }

  // Server returned error code
  if (responseCode != HTTP_CODE_OK) {
    Serial.printf("[WIFI] Server %d Response: %s\n", responseCode,
                  response.c_str());
    return false;
  }
```

```
  // Success
  Serial.println("[WIFI] Data sent to the server.");
  return true;
}

void processCapQueue(Queue* capCache, int capSize, char* endpoint) {
  int queueLength = capCache->getCount();
  for (int i = 0; i < queueLength; i++) {
    Serial.printf("[CACHE] retrieved oldest CAP. %d remaining.\n",
                  queueLength - i - 1);
    char cap[capSize];
    capCache->pop(cap);
    bool success = sendPostRequest(cap, endpoint);
    if (!success) {
      Serial.println("[CACHE] Request failed. Readding to cache.");
      capCache->push(cap);
    }
  }
}
```

FIRMWARE/src/wifi/ntp.cpp

```
#include <NTPClient.h>
#include <WiFiUdp.h>

#define TIME_ZONE ((11 * 3600) / 2)
#define TIMESTAMP_SIZE 50

WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP);

char formattedDate[50];

/**
 * Initialize NTP connection.
 * The WiFi must ve set-up at this point.
 * */
void initializeNtp() {
  Serial.println("[NTP] Starting NTP connection...");
  timeClient.begin();
  timeClient.setTimeOffset(TIME_ZONE);
  Serial.printf("[NTP] NTP connected in %d time zone.\n", TIME_ZONE);
}

/**
 * Update NTP time and set global variables
 * */
void updateTime() {
  Serial.println("[NTP] Retrieving timestamp via ntp...");
  while (!timeClient.update()) {
    timeClient.forceUpdate();
  }
  String date = timeClient.getFormattedDate();
  date.replace("Z", "+05:30");
  date.toCharArray(formattedDate, TIMESTAMP_SIZE);
  Serial.printf("[NTP] Retrieved timestamp: %s\n", formattedDate);
}
```

References

[1] https://fritzing.org/ - Fritzing tool was used to design the schematic diagrams of the prototype.