A Comparison of Power Consumption and Sampling Rate in Small-Scale IOT Devices

Lance Dinh, Sailesh Kadam, Katie Piens

Abstract

The goal of this project is to create a cost-effective small-scale Internet of Things (IOT) weather station and determine an optimal balance between the power consumption and sampling rate of the data collected. When designing a sensor-based device, there is a tradeoff between the amount of power supplied, including the cost and space allocation to store it, and the sampling rate at which data is collected over a period of time. Over a ten week time frame, a space-efficient, Arduino-controlled weather station was designed, created, and tested in order to compare the relationship between the power usage and sampling rate of the five selected sensors. The results of this testing show that when the sampling rate is increased to a higher frequency of data points being taken, the power consumption decreases at a logarithmic rate proportional to this.

Introduction

In the design process of a small-scale Internet of Things project, often the limiting factor is how to supply the amount of power required to enable the sensors to gather a sufficient amount of data to be useful in capturing patterns. Tradeoffs to overcome this issue include having a large power supply or battery pack in order to provide the necessary voltage and current output to the system, decreasing the amount of sensors used on the device, or decreasing the sampling rate at which data points are taken in order to conserve the power supplied. The purpose of this experiment is to determine the relationship between the sampling rate using an Arduino Uno system with five sensors collecting environmental data at specified intervals in order to determine the lifetime of four 1.5 V batteries connected in series. By understanding the tradeoffs of power consumption in a small-scale IOT device, future systems can be optimized in order to gather the most representative data while efficiently using the connected battery pack.

Design Process

The design of small-scale IOT weather station began with compiling a list of the types of data that might be useful and what sensors are compatible for each measurand. The chosen measurands for the weather station include an ambient temperature, barometric pressure, humidity, ambient light, and two raindrop sensors.

A schematic is shown below in Figure 1 with the arrangement of the sensors on top of the electronic enclosure.

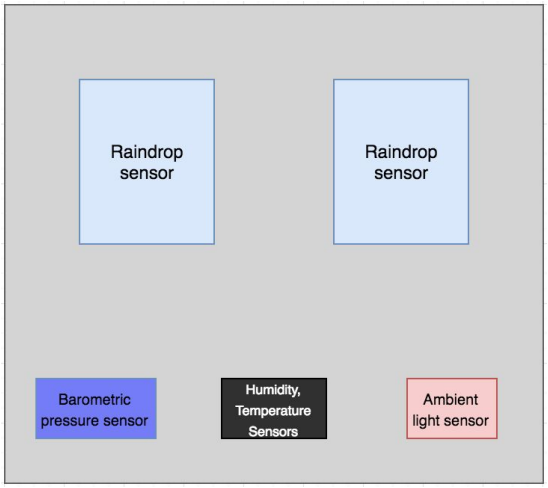


Figure 1. Schematic of small-scale IOT weather station sensors

Once the layout of the sensors was decided upon, the pinout diagrams for each sensor were used in order to establish the connections to provide each sensor with the proper input voltage.

Throughout the design process of a small-scale IOT weather station, various setbacks were encountered during the build phase. The first problem was finding a large enough power source in order to supply a sufficient voltage and current for the Arduino, sensors, and shields to be powered to collect the data. When the calculations were completed to find the minimum amount of power required for the system, it was found that a power source would need to be the size of a car battery in order to provide a high sampling rate for each sensor and power a bluetooth module in order to collect live data from the weather station. Since obtaining and using a car battery to power this small device was not feasible, in order to preserve battery, further measures were taken on looking for ways to reduce the power consumption of the sensors.

One way to reduce the power consumption is to utilize the sleep function for the Arduino controller was utilized in order to improve the energy efficiency of the system. Using a time step input, the Arduino will go into sleep mode for the allotted time set by the user, then wake up and

collect data from the sensors, then return to sleep mode in order to conserve power and reduce the usage. By adjusting the length of the sleep cycles of the Arduino, the duration the battery pack lasts will directly be affected.

Following this, the Arduino code was compiled in order to easily change the sampling rate of when the data would be gathered and loaded onto the microcontroller for each test while changing the time in which the Arduino would enter sleep mode.

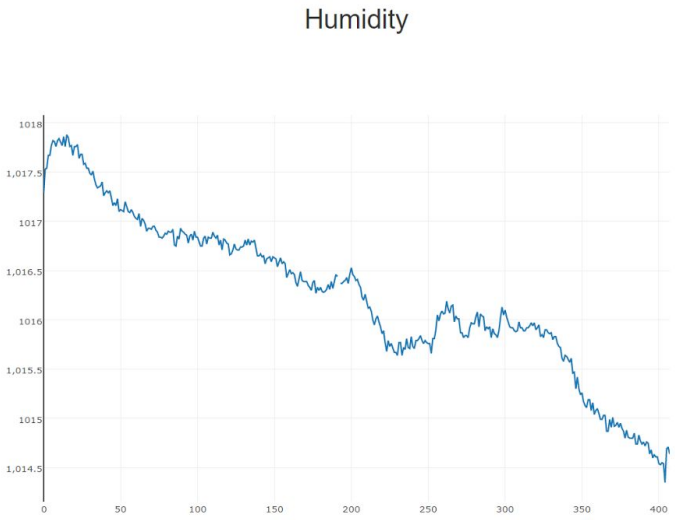
An additional setback that was encountered was the overall weatherproofing of the enclosure containing the electronic components. The first attempt at building a custom enclosure consisted of a 3D printed box to house the Arduino and the breadboard with the sensor connections. Once assembled, a water leakage test was conducted to ensure that water would not penetrate through the seams where the box was connected. During the testing, minor amounts of water were able to leak into the enclosure at the seams making this design unsuitable for meeting the requirements of a waterproof enclosure. The final design for the enclosure was to modify an existing airtight container by cutting out slots for the sensors on the lid of the container, then sealing the edges with hot glue in order to create a watertight seal protecting the electronic components.

The Design process for the web-app was trivial. Because we are gathering (non-live data), we thought it best to be able to upload csv files to a simple web app that essentially copied it into a database. Then from that database, we could display the desired data. I used golang for the backend because it is very nice and quick to get a good website up and running. I used sqlite3 for the database, since we knew the database wasn't going to be used extensively. For the frontend, a combination of Javascript, HTML, CSS, and Bootstrap were used. For plotting, a plugin called plot.ly. We used this because it was able to render big data sets really well. In addition we have a table displaying raw values.

Data

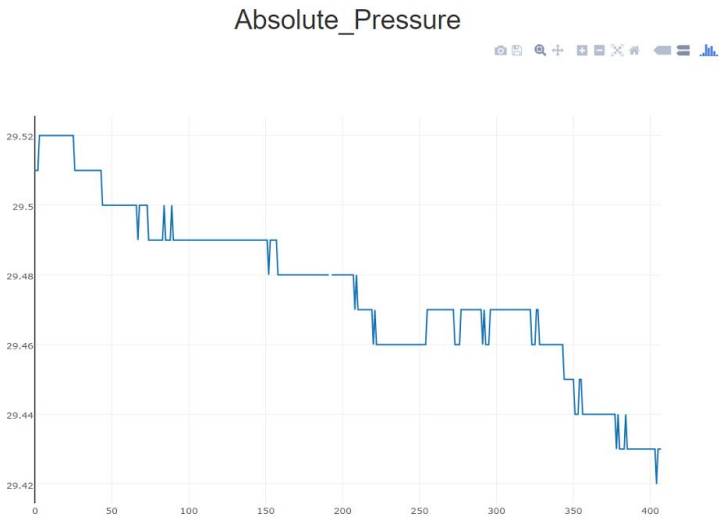
Select Data To View

Real Data.csv | Humidity



Select Data To View

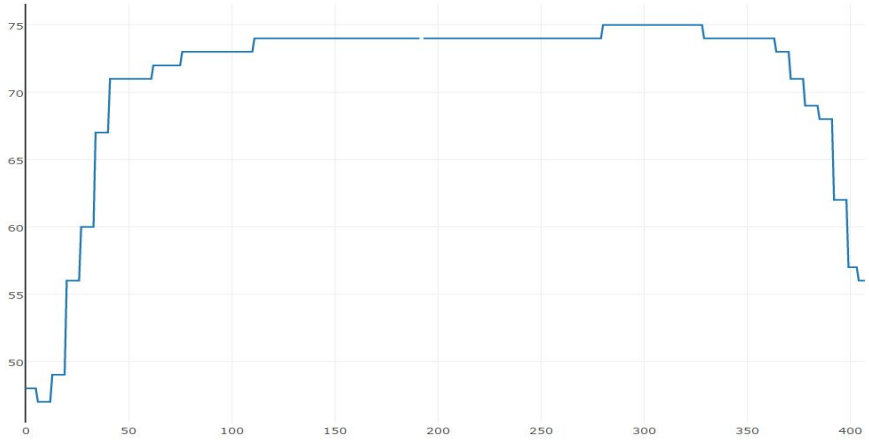
Real Data.csv | Abs Pressure



Raw Data

Time	Temp	Abs Pressure	Rel Pressure	Humidity	Rain Sensor 1	Rain Sensor 2
1	12.97	29.51	29.51	1017.29	68	19.31
2	7.74	29.51	29.51	1017.53	68	19.31
3	5.17	29.51	29.51	1017.54	68	19.31
4	3.62	29.52	29.52	1017.67	68	19.31
5	2.92	29.52	29.52	1017.67	68	19.31
6	2.06	29.52	29.52	1017.77	68	19.31
7	1.93	29.52	29.52	1017.82	69.8	20.38
8	1.47	29.52	29.52	1017.81	69.8	20.38
9	1.13	29.52	29.52	1017.76	69.8	20.38
10	0.84	29.52	29.52	1017.82	69.8	20.38
11	0.66	29.52	29.52	1017.84	69.8	20.38
12	0.54	29.52	29.52	1017.8	69.8	20.38
13	0.53	29.52	29.52	1017.77	69.8	20.38
14	0.44	29.52	29.52	1017.86	37.4	0.63
15	0.36	29.52	29.52	1017.76	37.4	0.63

Temp_F



The following data is taken from weather.com from the same day that we recorded data with our device. By comparing the hourly values with those recorded from the small-scale IOT device, the values show to be similar, while being no more than 5% difference between the data collected and the nominal conditions in the Terre Haute area.

Results and Discussion

The sample data, shown above, was taken at a sampling rate of one data point for each sensor every 40 seconds. By utilizing a higher sampling rate, more data points are taken within a given time providing a better estimate for the small changes in the weather. In order to optimize the balance between power and sampling rate of data, a minimum resolution for data must first be established. This can be found through testing data at various sampling rates and determining what the lowest sampling rate is that still provides an accurate representation for the weather in the area.

In order to check the accuracy of the weather station, the data collected was compared to real time results from local weather stations to ensure the data was reading accurate values. When compared to nominal values, all values collected were within 5% difference of the accepted Terre Haute, Indiana weather points. This provides assurance that the sensors were properly calibrated and that the data is useful for the area sampled.

A good point to mention is that with no data logger shield, the apparatus can run for approximately 38 hours. Temperature will affect this value, in that a lower temperature yields a lesser battery capacity. The opposite is true for higher temperatures. The data logger shield takes extra battery power, because it takes more power to write to the SD card. With the apparatus taking data and the data logger shield writing data as fast as they could, the battery lasted for about 20 hours. Changing the sampling rate to once a second (instead of once every tenth of a second which is our fastest sampling rate), increased the battery life to 29 hours. Raising it to once every 40 seconds raised the battery life to about 84 hours. Because we didn't have enough pins to use get time stamps from the data shield (because the timestamp function of the data logger shield requires use of two analog pins which we didn't have since we used it for the DHT11). The values for the times taken before battery life ended was based off the sampling rate and number of data points

We tried running the apparatus for a week to see how long it would run, but during the time an unknown force (most likely a raccoon or some small animal) in the woods where the apparatus was taking data bumped into it and shifted the batteries which caused the apparatus to lose power.

The results show that by increasing the sampling rate, the battery life will decrease as expected. This is shown in the tests comparing the overall life of the battery pack attached when changing from a fast sampling rate to a much slower one.

Lessons Learned & Test and Refine

A main lesson learned through this project was that the allocation of power for an independent system is a limiting factor in the amount of sensors used as well as the sampling rate in order to optimize the overall system in terms of data gathered and energy efficiency. When first determining the amount of power needed for a small system, several factors must be considered: the estimated time between switching power sources (ie. battery packs in this case), the numbers of sensors connected through the Arduino and the voltage requirements for each, and the sampling rate at which data is gathered for each sensor.

Another lesson learned in this project is the difficulty of waterproofing electronics. Since the purpose of this system is to be a small-scale weather station, it needs to be able to survive in various weather conditions, the most harmful being rain. If water were to infiltrate the electronics, it would damage the Arduino and the connections on the breadboard skewing the data or possibly breaking the microcontroller.

One important consideration when designing an independent system is the weight distribution of the components mounted onto the frame. The main electrical components of the system include an Arduino Uno, an SD card shield, a small breadboard to connect the sensors to their respective voltages, ground, and analog or digital inputs. The overall organization of these parts to equally distribute the weights was taken into consideration during the design phase to ensure the final device would be able to sit level to collect data.

Appendix

Arduino Code:

```
#include <Wire.h>
#include <DHT.h>
#include <LowPower.h>
#include <SFE_BMP180.h>
#include <SPI.h>
#include <SD.h>

#define DHTPIN 9
#define DHTTYPE DHT11
#define ALTITUDE 152.1

DHT dht(DHTPIN, DHTTYPE);
SFE_BMP180 pressure;

int rain1 = A0;
int rain2 = A1;
int light = A2;

int RainSense1 = 0;
int RainSense2 = 0;
int LightSense = 0;

const int chipSelect = 10;
File dataFile;

void setup() {
  Serial.begin(9600);
  dht.begin();
  pressure.begin();
```

```
  Serial.print("Initializing SD card...");
  // make sure that the default chip select pin is set to
  // output, even if you don't use it:
  pinMode(10, OUTPUT);


  // see if the card is present and can be initialized:
  if (!SD.begin(10)) {
    Serial.println("Card failed, or not present");
    // don't do anything more:
    while (1) ;
  }
  Serial.println("card initialized.");
  dataFile = SD.open("IOT.txt", FILE_WRITE);
  if (! dataFile) {
    Serial.println("error opening datalog.txt");
    // Wait forever since we can’t write data
    while (1) ;
  }

  SD.begin();

}

void loop() {
  char status;
  double T,P,p0,a;

  status = pressure.startTemperature();
  if (status != 0)
  {
    // Wait for the measurement to complete:
    delay(status);

    // Retrieve the completed temperature measurement:
    // Note that the measurement is stored in the variable T.
    // Function returns 1 if successful, 0 if failure.

    status = pressure.getTemperature(T);
```

```
  if (status != 0)
  {
    // Print out the measurement:
    //Serial.print("temperature: ");
    dataFile.print(T,2); //Temperature in Degrees C
    //dataFile.print(",");

    // Start a pressure measurement:
    // The parameter is the oversampling setting, from 0 to 3 (highest res, longest wait).
    // If request is successful, the number of ms to wait is returned.
    // If request is unsuccessful, 0 is returned.

    status = pressure.startPressure(3);
    if (status != 0)
    {
      // Wait for the measurement to complete:
      delay(status);

      // Retrieve the completed pressure measurement:
      // Note that the measurement is stored in the variable P.
      // Note also that the function requires the previous temperature measurement (T).
      // (If temperature is stable, you can do one temperature measurement for a number of
pressure measurements.)
      // Function returns 1 if successful, 0 if failure.

      status = pressure.getPressure(P,T);
      if (status != 0)
      {
        // Print out the measurement:
        //dataFile.print("absolute pressure: ");
        dataFile.print(P*0.0295333727,2); //Absolute pressure in inches of Hg
        //dataFile.print(",");
        //dataFile.print(" mb, ");
        dataFile.print(P*0.0295333727,2);
        //dataFile.println(" inHg");

        //Serial.print("absolute pressure: ");
        Serial.print(P*0.0295333727,2); //Absolute pressure in inches of Hg
        //Serial.print(",");
```

```
      //Serial.print(" mb, ");
      Serial.print(P*0.0295333727,2);
      //Serial.println(" inHg");

      // The pressure sensor returns abolute pressure, which varies with altitude.
      // To remove the effects of altitude, use the sealevel function and your current altitude.
      // This number is commonly used in weather reports.
      // Parameters: P = absolute pressure in mb, ALTITUDE = current altitude in m.
      // Result: p0 = sea-level compensated pressure in mb

      p0 = pressure.sealevel(P,ALTITUDE); // we're at 1655 meters (Boulder, CO)
      //dataFile.print("relative (sea-level) pressure: ");
      dataFile.print(p0,2);
      //dataFile.print(" mb, ");
      dataFile.print(p0*0.0295333727,2); //relative pressure in inches of Hg
      //dataFile.print(",");
      //dataFile.println(" inHg");

      //Serial.print("relative (sea-level) pressure: ");
      Serial.print(p0,2);
      //Serial.print(" mb, ");
      Serial.print(p0*0.0295333727,2); //relative pressure in inches of Hg
      //Serial.print(",");
      //Serial.println(" inHg");
    }
   }
  }
}

RainSense1 = analogRead(rain1);
RainSense2 = analogRead(rain2);
LightSense = analogRead(light);

//Serial.println(sensorValue,DEC);

float h = dht.readHumidity();
// Read temperature as Celsius (the default)
float t = dht.readTemperature();
// Read temperature as Fahrenheit (isFahrenheit = true)
```

```
float f = dht.readTemperature(true);

// Check if any reads failed and exit early (to try again).
if (isnan(h) || isnan(t) || isnan(f)) {
  dataFile.println("Failed to read from DHT sensor!");
  return;
}

// Compute heat index in Fahrenheit (the default)
float hif = dht.computeHeatIndex(f, h);
// Compute heat index in Celsius (isFahreheit = false)
float hic = dht.computeHeatIndex(t, h, false);

//dataFile.print("Humidity: ");
dataFile.print(h);
//dataFile.print(",");
//dataFile.print("Temperature: ");
dataFile.print(t);
//dataFile.print(",");
dataFile.print(f);
//dataFile.print(",");
//dataFile.print("Heat index: ");
dataFile.print(hic);
//dataFile.print(",");
dataFile.print(hif);
//dataFile.println(" *F");

dataFile.print("Rain 1: ");
dataFile.print(RainSense1);
dataFile.print(",");
dataFile.print("\nRain 2: ");
dataFile.print(RainSense2);
dataFile.print(",");

dataFile.print(LightSense);

Serial.print("Humidity: ");
Serial.print(h);
Serial.print(",");
```

```
  Serial.print("Temperature: ");
  Serial.print(t);
  Serial.print(",");
  Serial.print(f);
  Serial.print(",");
  Serial.print("Heat index: ");
  Serial.print(hic);
  Serial.print(",");
  Serial.print(hif);
  Serial.println(" *F");

  Serial.print("Rain 1: ");
  Serial.print(RainSense1);
  Serial.print(",");
  Serial.print("\nRain 2: ");
  Serial.print(RainSense2);
  Serial.print(",");

  Serial.print(LightSense);

  Serial.println();

  dataFile.flush();

//Enter Sleep cycle for multiples of 8 seconds, this is
  for(int i = 0; i < 5; i++){
    LowPower.idle(SLEEP_8S, ADC_ON,
TIMER2_OFF,TIMER1_OFF,TIMER0_OFF,SPI_OFF,USART0_OFF,TWI_OFF);
  }


}
```

Works Cited

"Australia: Buoy-Based Weather Station Campbell Scientific Documents Spectacular One-in-200-year Flood Event in Tropical Australia." *Australia: Buoy-Based Weather Station: Campbell Scientific Documents S...* N.p., n.d. Web. 10 Nov. 2016.

"Commonality Analysis for Floating Weather Stations." *Commonality Analysis for Floating Weather Stations*. N.p., n.d. Web. 10 Nov. 2016.

Greenemeier, Larry. "The Internet of Things Is Growing Faster Than the Ability to Defend It." *Scientific American*. N.p., 25 Oct. 2016. Web. 10 Nov. 2016.

"Hands-on-lab IoT Weather Station Using Windows 10." *Hackster.io*. N.p., n.d. Web. 10 Nov. 2016.

Ingenerare. "Easy IoT Weather Station with Multiple Sensors." *Instructables.com*. N.p., 21 May 2016. Web. 10 Nov. 2016.

"Make an IoT Weather Station with SAMI - ARTIK." *ARTIK*. N.p., 08 Sept. 2016. Web. 10 Nov. 2016.

Quinton, Amy. "Floating Weather Station Will Measure Water Evaporation." - *Capradio.org*. N.p., n.d. Web. 10 Nov. 2016.