The MAHIVE Project

Hub for IoT Sensors

University of Idaho Capstone 2020

Tristan Clawson & Jared Gradin

Table of Contents

Value Proposition

Hacking is an ever-increasing threat, and industry needs a way to defend itself physically and digitally. The MAHIVE project will merge cyberphysical and cybersecurity solutions into one platform. Using artificial intelligence and machine learning, MAHIVE will correlate physical and digital events in a way never seen before. Team TBD was tasked with creating the hub controller, which would link IoT devices, consolidate reports into a JSON/RDF format, encrypt them, and then securely send them to a MAHIVE server using advanced authentication and encryption.

Design Requirements

The project had three main requirements:

- The device must authenticate with the server using public key infrastructure (PKI)
- The device must format the sensor data into JSON.
- The device must encrypt the JSON and send it to the server for decryption.

Other design constraints included physical considerations:

- Design must be low-cost (goal for sensors: <$50).
    - No elaborate solutions that take large amounts of money to implement.
- Design must use off-the-shelf parts/boards.
    - Raspberry Pi, Arduino, etc.
    - No custom-designed boards or parts.
    - Utilize simple sensor parts readily available online.
- Sensors should ideally run off battery.
    - Board used for sensors should be built for low power consumption.
    - Device not constantly polling; use interrupts if possible.
- Sensor devices should be small and easily placed inside buildings.
    - Should not attract attention to prevent tampering.

Project Background

In the past, cyber security has been limited to the digital world, without a way to defend against physical attacks. While digital threats are still a relevant concern, special attention must also be paid to physical security of devices and data. If someone with enough knowledge or expertise were afforded physical access to a computer system, server, or data storage, they could cost an organization several thousands of dollars in damages.

The MAHIVE project has the goal of correlating physical and digital events into a system that can provide total security. It will utilize multiple elements to accomplish this. The main feature of the overall project is an AI-enabled server to correlate events together to identify an intruder or malicious actor. Our project is a smaller part; it covers the individual sensors that will act as the initial data collection.

Recent advances in microcontrollers and microprocessors now allow for sensors to be connected as IoT devices. This would be similar to a smart home setup, with a modular network of sensors reporting data to a single hub device. This hub would then relay all sensor data back to the server. Utilizing IoT devices like this would allow for a modular system that could be adapted to any number of buildings and situations.

Device Research

We conducted research of multiple microcontroller devices to determine which would best fit our needs. We ended up looking for one type of device to function as a hub and another to be used for sensors, due to the project's larger initial scope. The devices we looked at included the Arduino Uno, Arduino Mega, Espressif ESP32, Raspberry Pi Zero W, and Raspberry Pi 3.

The first device we researched was the Arduino Uno. At $11-19, it was reasonably priced. Its power consumption was relatively low, and documentation was plentiful. Numerous add-ons (shields) were available to expand functionality. However, its CPU resources were minimal when compared with other devices. In addition, either a Wi-Fi or ethernet shield would be needed to give it the basic functionality we needed. Adding these extra components would have driven up the power consumption and price too far.

Although a similar device to the Uno, the Arduino Mega packs more computational resources and could have been a viable option. The level of support was on par with the smaller Arduino, and encryption libraries could be run without much issue. However, its size was larger than we would have wanted for an IoT device. Its higher power consumption meant that it could not be run on battery. The fact that its price was high prevented this from being a viable option.

The ESP32 is a small, very low power microcontroller. This device has built-in bluetooth and Wi-Fi connectivity, as it was primarily designed with IoT devices in mind. It also had significantly more CPU resources than the Arduino Uno. It is explicitly supported by Amazon FreeRTOS and is able to integrate with AWS IoT Greengrass. Compared to the Arduino, the ESP32 has much less support and documentation; development would be more complicated. However, the fact that it was able to integrate with Greengrass drove us to choose this device for use as a sensor node.

The Raspberry Pi Zero W had a very reasonable price, especially when compared with the Raspberry Pi 3 or Arduino Mega. It packs all the functionality of the first-generation Raspberry Pi into a much smaller size. The Zero W model included built-in Wi-Fi as well. Almost all the open-source software available to the full-size Pi would run on the Pi Zero W. In spite of this, the device was still overkill as a sensor device, especially because it would not be able to run off battery for any length of time. Additionally, the Pi Zero W did not have the external I/O to be utilized as a hub device. It also did not support quite as many programs as the full-size Raspberry Pi. It occupied a weird middle ground and wasn't useful for either case.

For the hub device, we ended up choosing the Raspberry Pi 3. Compared to other boards, it had much greater processing power. Since it is more of a low-power computer than a microcontroller, it has much more software support and functionality. It supported many different libraries and programs. In addition, its popularity meant that a plethora of documentation existed online. It was able to run the AWS Greengrass Core software. As a hub device, the need for AC power is not an issue, so we decided it was the best option for the job.

IoT Platform Research

In addition to devices, we also researched potential IoT platforms to use. Using a pre-made platform would theoretically simplify development. Instead of spending time setting up authentication and encryption, development time and resources could be concentrated on the devices themselves. From our research, we were faced with the choice between AWS IoT Greengrass, Azure IoT Hub/Edge, and our own platform made from scratch.

The benefits of our own platform developed from scratch would have been in the realm of control. With our own platform, we would not be tied to a single third-party company, and no special authorization would be needed to use it. However, programming the authentication, authorization, and messaging would have been a very involved process. The project would have been consumed with getting communication and authentication to work properly. We decided that our priority was the sensors themselves, rather than just the communication infrastructure. We decided to use a third-party IoT platform, and our scope shifted to researching whether a third-party IoT platform would be viable for this project's purposes.

Azure IoT was a very popular IoT platform, with the benefit that the core edge service is open-source. It runs in the cloud and integrates with the IoT edge service. A Raspberry Pi could act as a middle-man, running an open-source runtime. The ESP32 would connect to the Azure cloud service through the Pi. The main reason we did not choose this platform was the lack of clear documentation. This platform had a lot of features, but the documentation was hard to understand and we struggled to get a clear grasp of what it could do and how well it would work for our purposes.

The last option we looked at was AWS IoT Greengrass. Greengrass was a feature under the massive umbrella of AWS services and technologies. It functions in a similar way to Azure IoT; a device running Greengrass Core software functions as a hub, facilitating communication between the cloud and FreeRTOS-enabled edge devices. Its documentation was more clear in explaining its purpose, and we were able to see how it would satisfy our 3 requirements. The only main concern we had was being tied to a third party company (Amazon), especially because the Greengrass Core software was not open source. But because it seemed simpler (on the surface) and more possible to implement, we decided to use it as our platform. Due to the research nature of our project, we were able to obtain permission to use Greengrass.

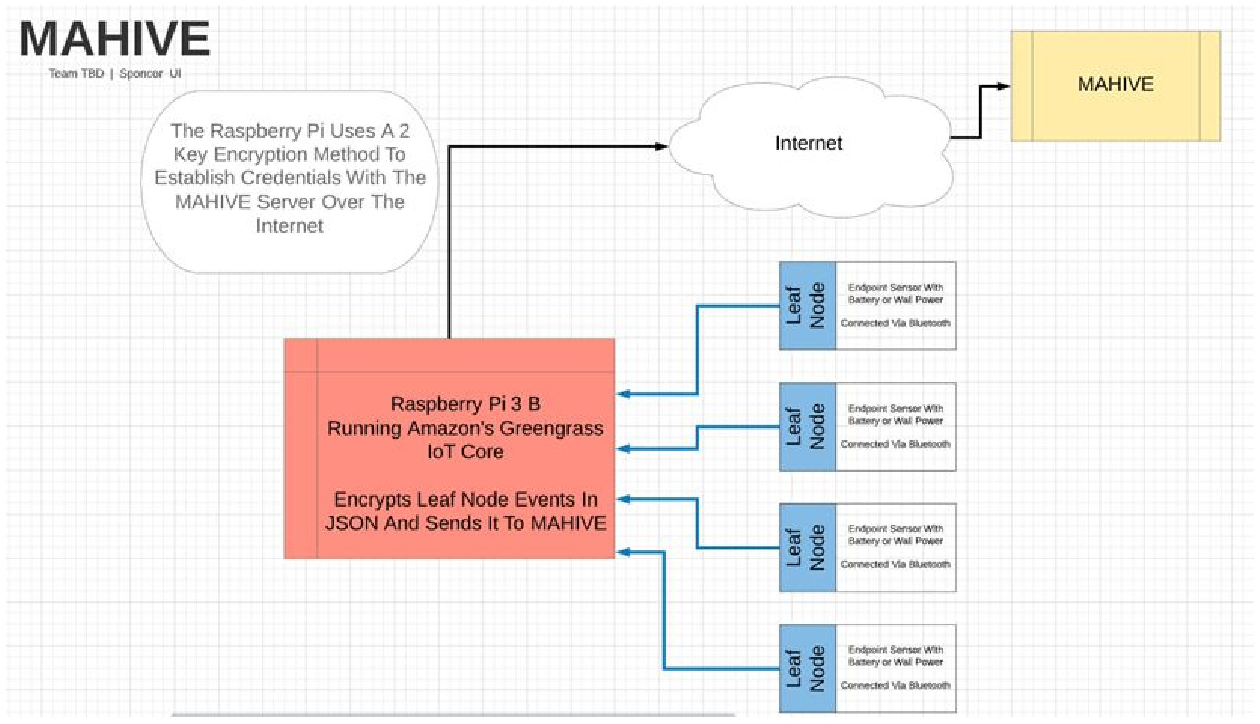
System Functionality

The goal of the project was to create a system that could be used in buildings, possibly with a few Hub devices per floor. The sensor devices could be placed throughout the rooms, communicating through the Hubs. This way, we could have a modular system that would be relatively easy to expand and cover a broader area. In addition, all the data would be formatted to JSON and encrypted before being sent up to the server (whatever form that took).

When the scope of the project changed, some of those goals changed. Having separate sensor devices is now a future goal, rather than one we have the ability to implement at this time. The ultimate goal of the project remains the same; however, in its current state the project is much different. Each Hub device would be confined to a single room, with wired sensors detecting doors opening, lights turning on, etc. To cover a building, numerous Hub devices will be needed to cover each room. The Hub device we have, although it does not currently route communications of IoT devices up to the cloud, still has that capability due to it running the Greengrass Core software.

When a sensor trips, a local Python program on the device logs the event into 2 files. The first, an archive, is saved once per day to provide an additional record of all sensor events. The second, a buffer, only holds the current sensor event. The locally deployed lambda function will extract the data from the buffer, if it is new data. The function then publishes the contents of the file to the AWS cloud, using a specific topic name. An endpoint can access this data by subscribing to the topic.

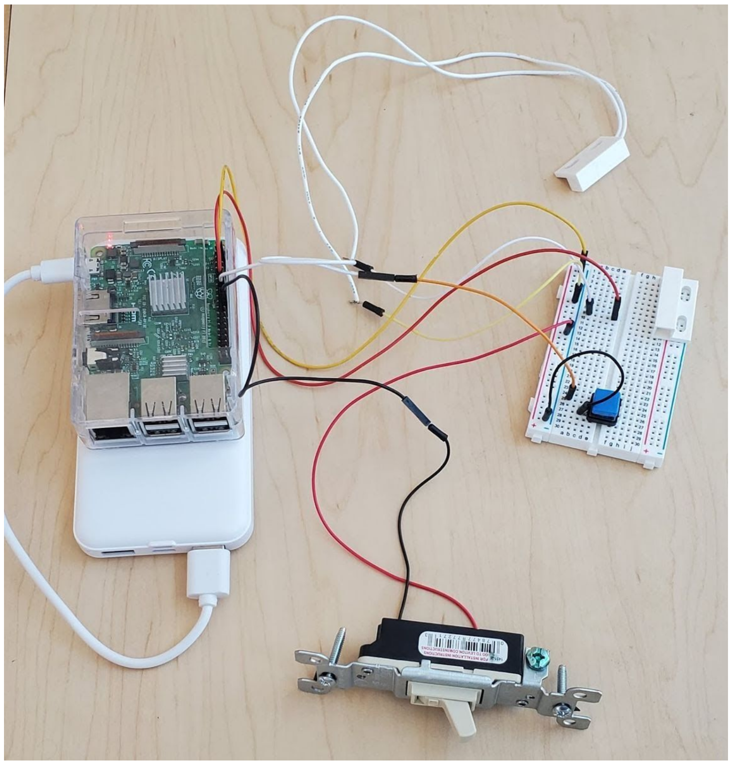
Logical Diagram


MAHIVE
Team TBD | Sponcor UI
The Raspberry Pi Uses A 2 Key Encryption Method To Establish Credentials With The MAHIVE Server Over The Internet
Internet
MAHIVE
Raspberry Pi 3 B Running Amazon's Greengrass IoT Core
Encrypts Leaf Node Events In JSON And Sends It To MAHIVE
Leaf Node
Endpoint Sensor With Battery or Wall Power
Connected Via Bluetooth
Leaf Node
Endpoint Sensor With Battery or Wall Power
Connected Via Bluetooth
Leaf Node
Endpoint Sensor With Battery or Wall Power
Connected Via Bluetooth
Leaf Node
Endpoint Sensor With Battery or Wall Power
Connected Via Bluetooth

Team TBD's original plan for the project is represented by this diagram. The Hub device, shown here in red, acts as a middle man between the sensors (shown here as Leaf Nodes) and the internet. The ultimate goal would be for this system to securely send data over the internet over to the MAHIVE server, which would be developed to use Machine Learning to perform data analysis.

Due to unforeseen circumstances changing our plans, the Leaf Nodes were pushed aside and the focus shifted to just the Hub device. The main change to the logical diagram would be that the leaf nodes are replaced by individual sensor hardware, wired straight to the GPIO of the Hub. In addition, since it was out of the scope of this project, the Hub device sends data to the AWS cloud, with the potential for it to be routed to a service for storage or processing. At the moment no such infrastructure is yet implemented.

Physical Representation



Clockwise from left: battery pack, hub device, mock door sensor, kill switch, and mock light switch/sensor

The tech demo shows a basic representation of the final design. Both sensors, along with the kill switch, are wired to the GPIO of the Pi. This was in lieu of individual IoT devices being used as sensors, after complications in project development prevented that from happening. A local python program runs at boot to take in data from the sensors as they trip. The battery pack was used for demo purposes, but in a finished product it would be replaced by either AC power or power over Ethernet (POE). Although this configuration is more simplistic than originally planned, the Hub device can still be used with IoT devices running Amazon FreeRTOS.

Project Learning

AWS Greengrass Documentation Home: https://docs.aws.amazon.com/greengrass/index.html

AWS Greengrass Getting Started Guide: https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-gs.html

Amazon FreeRTOS Getting Started Guide: https://aws.amazon.com/freertos/getting-started/

Raspberry Pi 3 GPIO Documentation: https://www.raspberrypi.org/documentation/usage/gpio/

Once research into platforms and devices was finished, and plans for what to use were finalized, the next step was to learn the ins and outs of the platform. The first step we took was running through the AWS Greengrass getting started guide. This tutorial walks through the steps to setup AWS Greengrass core software on the Hub device. It covers installing on the Raspberry Pi, configuring the Greengrass group on the AWS console, and starting the Greengrass Core program.

The Getting Started Guide also walks through basic setup of lambda functions on the Raspberry Pi. Lambda functions get deployed locally on the device to perform various functionality. In our project, we use a lambda function to loop through various files in the Raspberry Pi’s filesystem and relay them to the AWS cloud. Setup for basic lambda functions, as well as more complex ones, is covered in the Getting Started Guide.

Our goal was to get basic demos up and running, and modify the working code to perform our needed functionality. Once the Greengrass Core software was set up on the Hub device, we decided to set up Amazon FreeRTOS for use as a sensor device. The Amazon FreeRTOS Getting Started Guide walks through a very basic demo setup of Amazon FreeRTOS on an ESP32 microcontroller. It covers the setup of the toolchain and the development environment for compiling and flashing Amazon FreeRTOS to the ESP32 device. The tutorial sets up a demo application which discovers the Hub device and sends “Hello World” messages up to the AWS cloud.

The GPIO of the Raspberry Pi was used in our final iteration, as it was where the sensors and kill switch were wired in. The guide showed us the various GPIO pins to use for our purposes. We needed the GND pin, as well as pins 23, 24, and 25. The GPIO is controlled and interfaced with via the Rpi.GPIO Python library. This was then used in our code to program interrupts for each GPIO input.

Lambda Function Code

```
# TEAMTBD_function01.py
#
# This program is the code for the lambda function. It retrieves the timestamp, name, location, etc. from each of the
buffer files
# and publishes it to the given topic. It has checks to ensure that only new data is published (no repeated values)
and that the function
# will only publish to the topic if the file is not blank.
# It is currently configured to update for new data every second. This could be changed to be longer or shorter
depending on the needed update frequency.

import logging
import platform
import sys
import os
from threading import Timer
import greengrasssdk

# Setup logging to stdout
logger = logging.getLogger(__name__)
logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)

# Creating a greengrass core sdk client
client = greengrasssdk.client("iot-data")

# Retrieving platform information to send from Greengrass Core
my_platform = platform.platform()

# function for opening the buffer file and retrieve the timestamp information
def openSensorFile(filename):
    with open(filename, 'r') as log:
        data = log.read()
    return data

# main function that does most of the work
def greengrass_hello_world_run():
    try:
                  # list of file names to read from
        sensorFiles = ["/home/pi/sensor01_buffer.txt", "/home/pi/sensor02_buffer.txt"]

                  # read data from each file
        data1 = openSensorFile(sensorFiles[0])
        data2 = openSensorFile(sensorFiles[1])

                  # publish to topic only if the file is not empty and the file data has changed since the last read
        if data1 != greengrass_hello_world_run.initial_value_0:
            if data1 != greengrass_hello_world_run.previous_value_1:
                client.publish(topic="teamtbd/hub", queueFullPolicy="AllOrException", payload=data1)
        if data2 != greengrass_hello_world_run.initial_value_0:
            if data2 != greengrass_hello_world_run.previous_value_2:
                client.publish(topic="teamtbd/hub", queueFullPolicy="AllOrException", payload=data2)

                  # assign the new data values to the previous_value variables as a check for the next read
        greengrass_hello_world_run.previous_value_1 = data1
        greengrass_hello_world_run.previous_value_2 = data2

         # if any exception occurs, write an error to the logger
    except Exception as e:
        logger.error("Failed to publish message: " + repr(e))

    # asynchronously schedule this function to be run again in 5 seconds
    Timer(1, greengrass_hello_world_run).start()


# main code
#
# initialize the previous_value and initial_value variables to a blank string
# so that the lambda function will not fail when starting with a blank file
greengrass_hello_world_run.previous_value_1 = ''
greengrass_hello_world_run.previous_value_2 = ''
greengrass_hello_world_run.initial_value_0 = ''

# start executing the function above
greengrass_hello_world_run()


# this is a dummy handler and will not be invoked
# instead, the code above will be executed in an infinite loop
def function_handler(event, context):
    return
```

Sensor Code

```
# TEAMTBD_sensors.py
# This program runs locally on the Raspberry Pi.
#
# Each sensor is wired to the GPIO, and every sensor event triggers an interrupt.
# the interrupt handler will fill in an entry to an archive file, as a backup archive.
# it will also fill an entry to a buffer file, which will be retrieved by the lambda
# function deployed on the AWS platform.

import RPi.GPIO as GPIO
from datetime import datetime
from datetime import date
from threading import Timer
import os

# initialize GPIO pins 23, 24, and 25
pin_list = [23, 24, 25]
# sensor number in a list
sensor_list = ["sensor01", "sensor02"]
# unique name for each sensor
sensor_names = ["doorsw_01", "lightsw_01"]

# set up all the GPIO pins that will be used
def gpio_setup(list):
        GPIO.setmode(GPIO.BCM)
        # loop through list of GPIO pins and initialize
        for number in list:
                GPIO.setup(number, GPIO.IN, pull_up_down=GPIO.PUD_UP)

# rename each archive file once per day so that offline records can be kept
def renameArchive(sensorList):
        datetimeNow = datetime.now()
        # get a string of the timestamp for naming the file
        timestampStr = datetimeNow.strftime("%Y%b%d%H%M%S")

        # rename each sensor's archive
        # the sensor will create a new blank archive once the existing file is renamed
        for sensor in sensorList:
                archiveStr = sensor + "_archive.txt"
                # the archive file will only be renamed if it is not empty, prevents extra file from being created
at startup
                if(os.path.getsize(archiveStr) > 0):
                        newArchiveStr = timestampStr + "_archive_" + sensor + ".txt"
                        os.rename(archiveStr, newArchiveStr)

# set the timer for renaming the archive file once per day
```

```
def timer_setup():
        # current time and day
        xTime = datetime.today()
        # next day at 1am
        yTime = xTime.replace(day=xTime.day+1, hour=1, minute=0, second=0, microsecond=0)
        # time difference
        delta_t=yTime-xTime

        secs=delta_t.seconds+1

        # set timer for the remaining amount of seconds
        # when time is up, renameArchive function is called for the list of sensors
        t = Timer(secs, renameArchive(sensor_list))
        t.start

# set up the buffer and archive files for each sensor
def sensor_setup(sensorList):
        for sensor in sensorList:
                bufferStr = sensor + "_buffer.txt"
                archiveStr = sensor + "_archive.txt"
                # quickly open and close each file to ensure both files are empty
                f0 = open(bufferStr, 'w').close()
                f1 = open(archiveStr, 'w').close()

# callback functions
# these could potentially be merged into one function, depending on whether the callback function can handle extra
arguments
# many extra arguments would be needed for sensor name, location, etc.

# callback function for first sensor
def sensor01_callback(channel):
        time = datetime.now()
        timeStr = time.strftime("%m/%d/%Y, %H:%M:%S")
        # clear buffer file so that only one entry exists in the file at a time
        f = open('sensor01_buffer.txt', 'w').close()
        # write the entry into each file
        with open('sensor01_buffer.txt', 'a') as file1_0:
                file1_0.write("Sensor: ")
                file1_0.write(sensor_names[0])
                file1_0.write("\nTimestamp: ")
                file1_0.write(timeStr)
        with open('sensor01_archive.txt', 'a') as file1_1:
                file1_1.write("Sensor: ")
                file1_1.write(sensor_names[0])
                file1_1.write("\nTimestamp: ")
                file1_1.write(timeStr)
                file1_1.write('\n\n')
```

```
# callback function for second sensor
def sensor02_callback(channel):
        time = datetime.now()
        timeStr = time.strftime("%m/%d/%Y, %H:%M:%S")
        # clear buffer file so that only one entry exists in the file at a time
        f = open('sensor02_buffer.txt', 'w').close()
        # write the entry into each file
        with open('sensor02_buffer.txt', 'a') as file2_0:
                file2_0.write("Sensor: ")
                file2_0.write(sensor_names[1])
                file2_0.write("\nTimestamp: ")
                file2_0.write(timeStr)
        with open('sensor02_archive.txt', 'a') as file2_1:
                file2_1.write("Sensor: ")
                file2_1.write(sensor_names[1])
                file2_1.write("\nTimestamp: ")
                file2_1.write(timeStr)
                file2_1.write('\n\n')

# main code
# set up GPIO
gpio_setup(list=pin_list)
# set up sensors
sensor_setup(sensorList=sensor_list)
# set up the timer
timer_setup()

# add interrupt handlers for each sensor, with 300ms debounce
GPIO.add_event_detect(25, GPIO.FALLING, callback=sensor02_callback, bouncetime=300)
GPIO.add_event_detect(23, GPIO.FALLING, callback=sensor01_callback, bouncetime=300)

try:
        # this part loops infinitely waiting for kill button to be pressed
        while True:
                # program will wait for falling edge of kill button
                # the interrupts will interrupt this loop and go to one of the callback functions
                GPIO.wait_for_edge(24, GPIO.FALLING)
                # if button is pressed, program will stop
                print("stopping program...")

# clean up GPIO if ctrl-C is pressed on the terminal
except KeyboardInterrupt:
        GPIO.cleanup()
# clean up GPIO if program ends via kill button
GPIO.cleanup()
```

Bill of Materials

Parts Used in Tech Demo				
Part	**Description**	**Quantity**	**Source**	**Total Cost**
Raspberry Pi	Hub Device	1	Amazon	$38.98
Raspberry Pi Case	Case for Hub Device	1	Amazon	$6.50
SD Card	Boot drive, flashed with Raspbian	1	Amazon	$5.79
Battery Pack	Power for Hub Device	1	Amazon	$10.00
Door Sensor	Sensor #1	1	Adafruit	$3.95
Breadboard	Hold the kill switch	1	Amazon	$4.10
Light Switch	Sensor #2	1	Moscow Building Supply	$2.10
			Total	$71.32

Future Work

The most obvious area for future expansion would be in the realm of sensors. In the current iteration of the project, the sensors are not on individual IoT-connected devices. Currently, this Hub device will allow for that, but it is not fully implemented. One of the first ways to expand would be to add this functionality and make the sensors completely wireless.

Another area of expansion would be data storage. The project currently does not have a method to store the data in the cloud, as we were unable to get quite that far. AWS allows the data to be streamed to the cloud for data analytics, so that could be an option. Another route for expansion would entail creating some sort of endpoint server as a stand-in for a future MAHIVE server.