Engineering Notebook

Robust, Lora-based Mesh Network - Solution for Wildfire Detection and Growth Forecast through Custom Prediction Algorithm

D-55: Shine Chang, Bhargav Eranki, Matthew Chen

Table of Contents

Motivations	**3**
Resources	**4**
Detailed Finances	**5**
Operation Overview:	**6**
Meeting #1 - Brainstorm & General Design	**7**
Meeting #2 - Basic Transmitter/Receiver	**11**
Meeting #3 - Starting the API	**13**
Meeting #4 - Unified Transceiver	**17**
Meeting #5 - Package as Library	**22**
Meeting #6 - Implementing Gateways	**24**
Meeting #7 - Implementing the API	**24**
Meeting #8 - Adding the Controllers	**27**
Meeting #9 - Sensor Integration	**29**
Meeting #10 - Optimization via Bits	**31**
Meeting #11 - AES Encryption	**33**
Meeting #12 - Server and Database	**35**
Meeting #13 - Generation of Testing Data	**37**
Meeting #14 - Interface Development and Testing	**39**
Meeting #15 - Client Interface Draft	**41**
Meeting #16 - Integration of Google Maps with React	**43**
Meeting #16 - Handling Larger Client Interface	**45**
Meeting #17 - Graphically visualizing History Data	**48**
Meeting #18 - Hardware Packaging and Analyzer	**52**
Meeting #19 - Interface and Hardware Package	**54**
Meeting #20 - Asynchronous Web Scraping	**56**
Meeting #21 - Correlation Matrix	**58**
Meeting #22 - Support Vector Regression Model	**61**
Meeting #23 - Dynamically Reloading Components	**64**
Meeting #24 - Re-evaluating Decision Model & Integrating w/ Client	**67**
Final Takeaways/Future Plans	**70**

Motivations

The ever more frequent natural disasters are harmful tragedies that develop from an unfortunate environmental coincidence. Some disasters have quantifiable signs that can be compiled into a measure of risk. Though the sensor technologies exist, there lacks a system to connect a large network of sensors to survey the environment. The project aims to develop a software framework for IoT sensor networks to be used in environmental surveillance. It presents a networking protocol capable of maintaining a mesh topology, allowing for a greater covered area.

Past IoT solutions lacked an extensible topology and a long-ranged radio, often only used to cover small areas. Most IoT are not designed to span very large areas – their architectures typically only implement a hub-and-spoke topology, limiting the network diameter by the radio range. Past radio modules typically have a range-to-power tradeoff, and as such most IoT solutions do not use modules with great peer-to-peer range. Contemporary techniques in the detection of wildfires – such as aerial, visual, or infrared – require the fire to have already grown considerably.

Resources

All relevant Github/repositories:

- Full repository for API/frontend:
  - https://github.com/beranki/loRAFire
- Repositories for low-level/hardware code:
  - https://github.com/PaddingProductions/LoRaNet
  - https://github.com/PaddingProductions/LoRaNetLib

Detailed Finances

Component	**Per Unit**
Arduino Nano	$0.50
ESP-8266	$1.50
Ra-02 LoRa Module	$2.00
DHT-22 Temperature & Humidity Sensor	$0.80
MQ-2 Gas Sensor	$1.00
Packaging Costs (PCB, Injection)	~$15.00

Hypothetical Deployment Scenario: 10km^2, 500m/node, 1km/gateway

Component	**Per Unit**	**Quantity**	**Price**
Gateways	$18.50	40	$740
Nodes	$19.30	400	$7,720
Cumulative			**$8,460**

Operation Overview:

Set-up on the hardware side involves only configuration; this includes things such as LoRa channel, frequency, and WiFi credentials. During the deployment of the nodes, the position – in longitude-latitude coordinates – must be recorded to be entered in the database.
As for the server-side software, a MongoDB cluster must be available for usage and the credentials must be passed to the server's secrets file. As a Node.js application, this server can be easily deployed on most server-side environments. Note that the server does need to be exposed on at least one port for serving the interface and API.

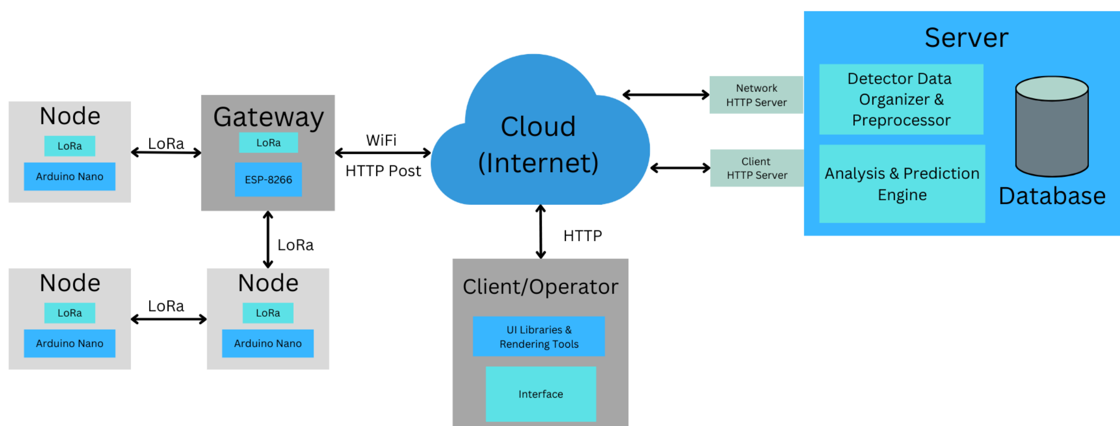
Meeting #1 - Brainstorm & General Design

Goals

- Define General Architecture: Software Modules, Hardware Components
- Secure Sponsor
- Order Parts

Meeting Notes:

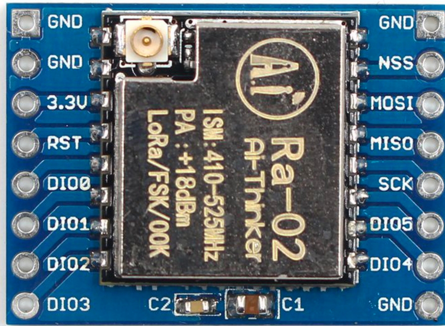
Settling on the idea of using an Ad-hoc sensor network to combat wildfires, we began drawing up the architecture. We began our research individually, analyzing current strategies in monitoring wildfires and reviewing past implementations of IoT technologies in the field.



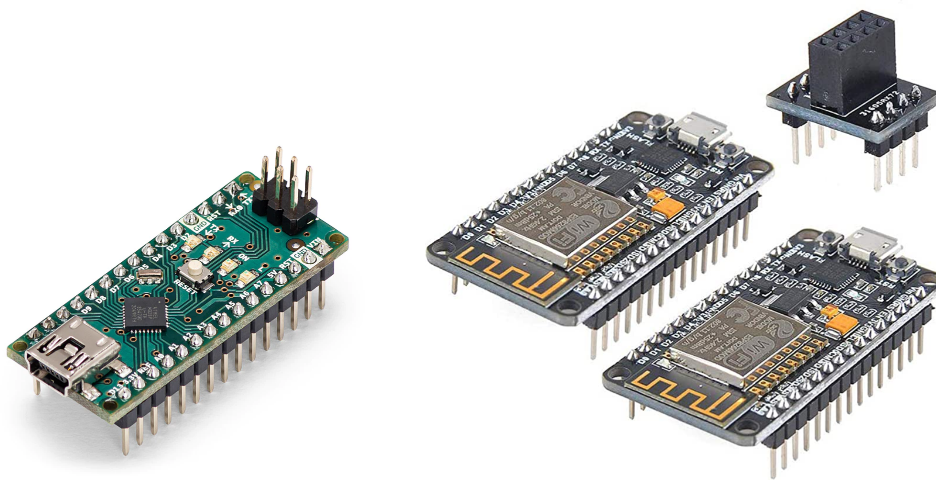
To make an IoT solution practical, it must be compact, power-efficient, and can maintain a sufficiently long peer-to-peer range. We also want to minimize the per-unit cost to further incentivise our solution.

We've already identified a feasible radio system to be used in the project, namely LoRa modulation. This new modulation, dubbed the "chirp modulation," has superb performance in both range and battery efficiency. The protocol boasts a peer-to-peer change of 10 kilometers in rural areas and 3 kilometers in dense or

urban areas. There are also existing LoRa modules that interact via standard interfaces, namely UART, SPI, and I2C. We settled with an SPI module, as there are existing Arduino libraries capable of handling this interface.

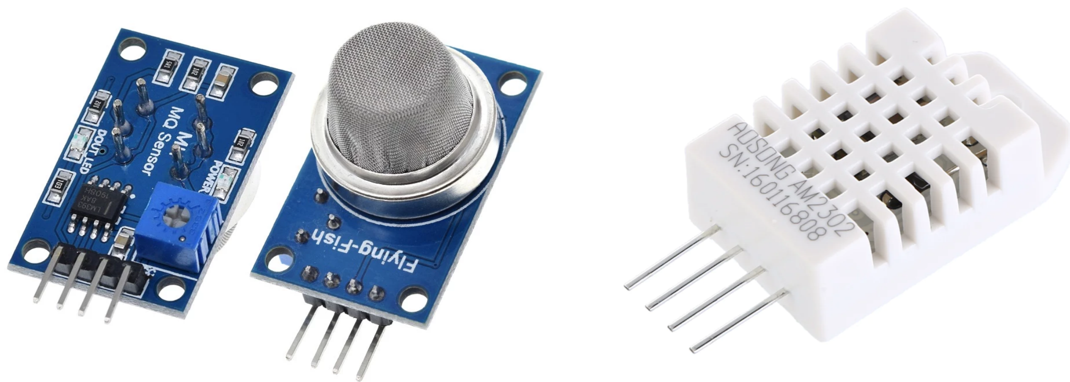


We chose to use Arduinos as our processor platform, specifically using the Arduino Nano as the microcontrollers for the sensor nodes. The Nano's dimensions are 18mm x 45mm, and being a popular and well-documented developing platform, would be a great fit for the project. The ESP-8266 was to be used for the gateways, as it could be programmed in a similar fashion as the Arduino while having WiFi-capabilities.



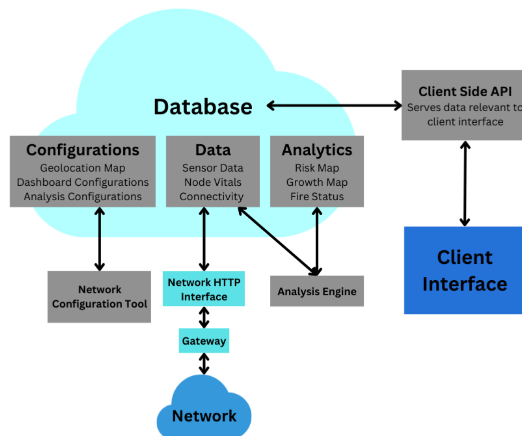
With the critical processing hardware secured, the next order of business was to secure some relevant sensors. As a draft, we selected the DHT temperature and humidity sensor and the MQ2 gas sensor. These will be sufficient to demonstrate the networking abilities of the network while also maintaining relevance to the topic of wildfires.

MQ-2



A draft of the database schema and client interface were also developed. The project was partitioned between the three developers, and we individually looked into the implementation of our assigned modules. The research plan was also drafted during this meeting.

Specifications were tough to write for our project. Beyond the physical nodes, the majority of our project was software, for which specifications are difficult to design. For the hardware specifications, we settled on some estimates for the size, cost, and range of the physical nodes. As for the software, we made a list of features for the user interface and a rough schema of the server-side API.



The next step was to design tests for our system to ensure functionality. We came up with two tests for the network. The first was a simple range test, to identify the maximum peer-to-peer distance between nodes and ensure that this range was within specifications. The second is a topology test, to demonstrate that the network was indeed capable of maintaining a mesh topology. This would be done by blocking set addresses to artificially create disconnections and a set topology.

Takeaways:

- Complete project architecture
- Secured hardware components
- Software model drafted

Meeting #2 - Basic Transmitter/Receiver

Goals:

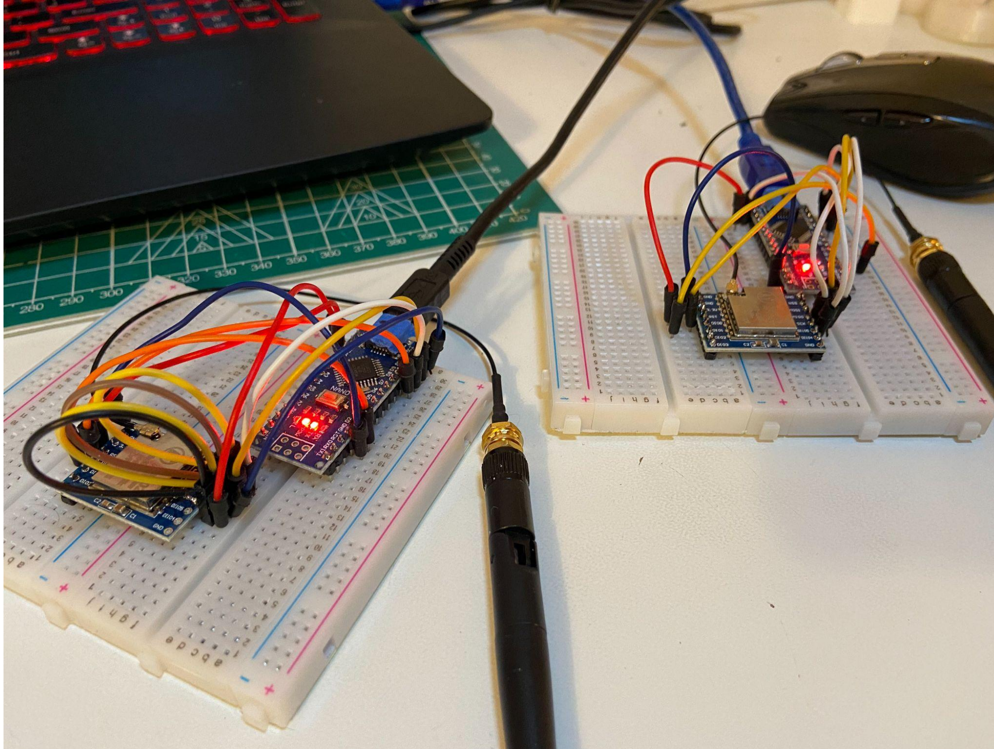
- Get familiar with LoRa & Arduinos

Meeting Notes:

Development began by starting with a simple program, slowly implementing more complex features to approach the development iteratively. This technique eases the debugging processes, as the issues can be traced with confidence to the newly added feature. The development iterations are as follows:

- Ping-ing Transmitter-Receiver Pair
- Ping-ing Transmitter & Pong-ing Receiver
- Callback implementation of Transmitter & Receiver

We choose to work towards a callback-based implementation to optimize power consumption, as the naive approach would've required constant polling on the radio module.



Though the duplex transmission did see some packets, there was an unacceptably low packet retention rate. After looking through online forums and testing the different proposed solutions, we were able to attain a near-perfect packet retention rate. The solution was to add a delay between mode switches of the LoRa module (receiving to transmitting, vise versa).

Takeaways:

- Now familiar with programming on the Arduino platform & interfacing with LoRa
- Complete Call-back Code
- Began development on server-side programs

Meeting #3 - Starting the API

Goals:

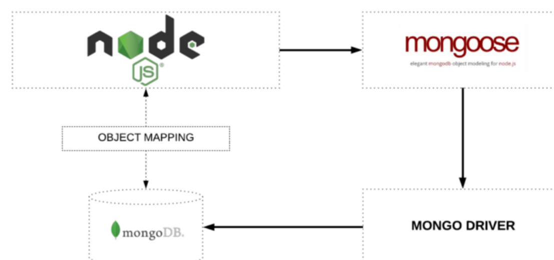
- Create the MongoDB database
- Set up the project structure for the API

Meeting Notes:

While work on the nodes were underway, we also began developing the server-side software. This includes things like a network-facing API, interface-facing API, and the database interface. We had a solid design and concept of the architecture we are aiming for, so assuming the design does not change in any major ways, we can develop the server and the network independently, then simply connect the two at the end of the project. This is to minimize development time and make use of all engineers on the team.

The server was to be written in Node.js, using the Express.js library. The database of choice was MongoDB, a convenient database for development, which has Mongoose as its Node.js interfacing library.

NODE_HISTORY	NODE	GATEWAY
Id	Id	Id
Table	Time	Location
Field	Location	Battery_Level
NodeId	Battery_Level	other important vitals..?
OldValue	GatewayId	
NewValue	HumiditySensorStats { stuff... }	
TimeAdded	Smoke_Sensor_Id { stuff...}	



To set up a database for the eventual data that we were going to display, we set up an account with MongoDB to store information. We stored the connection string in an environment file to keep it safe.

To structure the project, we decided to use Express.js for the API. We situated the API on localhost port 5000. To separate the future frontend from the interface, we decided on the route localhost:5000/api for the interface and localhost:5000 for the website.

Finally, we created folders for the models that describe the data, controllers for the get, post, put, and delete requests, and routers to create the actual routes.

```
var createError = require('http-errors');
var express = require('express');
var mongoose = require('mongoose');

var app = express();

const indexRouter = require('./routes/index');
const apiRouter = require('./routes/api');

require('dotenv').config();

// Setting up database connection
var mongoDB = process.env.DATABASE_URI;
mongoose.connect(mongoDB, {useNewUrlParser: true, useUnifiedTopology:
true});
mongoose.Promise = global.Promise;

var db = mongoose.connection;
```

```
db.on('connection', () => console.log("Successfully connected"));
db.on('error', () => console.error.bind(console, 'MongoDB connection
error'));

// Set render engine to EJS
app.set("view engine", "ejs");

// Validate the JSON data
app.use(express.json());


// Set up routes
app.use('/', indexRouter);
app.use('/api/', apiRouter);


// catch 404 and forward to error handler
app.use( (req, res, next) => next(createError(404)) );

// error handler
app.use((err, req, res, next) => {
    // set locals, only providing error in development
    res.locals.message = err.message;
    res.locals.error = req.app.get('env') === 'development' ? err :
{};
    // render the error page
    res.status(err.status || 500);
    res.json({
        message: err.message,
        error: err
```

```
    });
});



module.exports = app;
```

Takeaways:

- Completed the general structure for the API

- Connect an Express app to a mongoDB database

Meeting #4 - Unified Transceiver

Goals:

- Bridge the transmitter and receiver code

Meeting Notes:

The next step in development was to merge the transmitter and receiver code to form a transceiver program. This would later be evolved into our node program, as each node needs both transmitting and receiving capabilities.

The development target for this "transceiver" program was to be able to forward packets from peers to other peers. This would effectively be the proof-of-concept for our mesh network protocol. We again approached the development of the transceiver code iteratively, testing for the packet retention rate at every iteration.

```
/*
  Note on this code:
    This is the code for a Ping-Pong node, meaning each node will
ping the peer, expecting a pong/ACK packet.
  The purpose of this code is to assess the consistency of a Duplex
Transmission protocol, identifying the 'right' way to program such a
system.

  Implementation:
    An onReceive callback function will push an incoming 'ping' MSG
into a queue. The queue will be processed in the main loop,
  sending a 'pong' for every 'ping'. The program will also record the
packet loss rate.
*/
#include <SPI.h>
```

```
#include <LoRa.h>

#define ...

String queue[QUEUE_SIZE];
int queueCnt = 0;

void setup() {...}

int counter = 0;
int successes = 0;
int tick = 0;

void loop() {
  if (!tick --) {
    // Log success %
    if (counter % 10 == 0 ) {
      Serial.println("=== Packet Report, Acknowledged / Sent: " +
String(successes, DEC) +"/"+ String(counter,DEC)
        + " (" + (double) successes / counter + ") ===");
    }
    String msg = "ping no." + String(counter++, DEC);
    delay(100);

    LoRa.beginPacket();
    LoRa.print(msg);
    LoRa.endPacket();

    Serial.print("Sent ping, '" + msg + "'\n");
    delay(100);
    LoRa.receive();
```

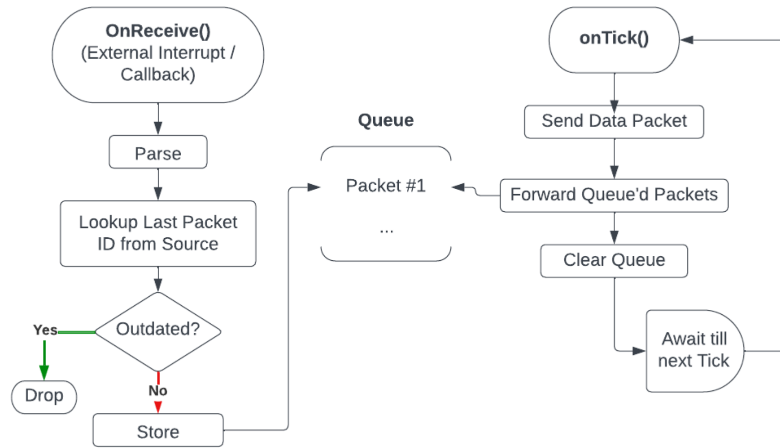
```
    tick = (TX_INTERVAL + random(0, RANDOM_RANGE)) / TICK_INTERVAL;
  }
  // Check & process msg queue.
  if (queueCnt)
    delay(100);
  for (int i=0; i<queueCnt; i++) {
    String ping = queue[i];
    String numStr = "";
    for (int i = ping.lastIndexOf("no.") + 3; i<ping.length(); i++)
      numStr += ping[i];
    int number = numStr.toInt();

    // Respond pong
    String msg = "pong no." + String(number, DEC);

    LoRa.beginPacket();
    LoRa.print(msg);
    LoRa.endPacket();
  }
  if (queueCnt) {
    queueCnt = 0;
    delay(100);
    LoRa.receive();
  }
  delay(TICK_INTERVAL);
}

void onReceive(int packetSize)
{
  // Store MSG in a queue.
  String msg = "";
  while (LoRa.available())
```

```
      msg += (char)LoRa.read();
    // If the queue is full, drop.
    if (queueCnt >= QUEUE_SIZE) {
      Serial.println("Queue full, dropping.");
      return;
    }
    // If is ping
    if (msg.lastIndexOf("ping") != -1) {
      queue[queueCnt++] = msg;
    } else { // If pong
      Serial.println("Got pong: " + msg + "'.");
      successes ++;
    }
  }
```

An ID-system had to be implemented at this stage. This system would allow nodes to track packets and prevent redundant forwarding. For ease of development, the packets were sent in human-legible string format, with the data represented as comma-separated values. The forwarding protocol works by maintaining a table mapping peer ID's to the ID of the last packet received. The packet ID would serve as a time-stamp-like system, ensuring that a packet with a greater ID means it was sent at a later time. With this table the system would be able to identify if a packet is new, and forward accordingly.

We also worked on a queue-based forwarding implementation. We opted for this design since switching between receiving and transmitting mode requires some delays, and might lead to some packet losses during this time. The queueing works by storing new packets it receives and staged for forwarding, transmitting it all at a set interval. We also added randomness in the forwarding intervals, to decrease the

chances of the “bursts” of forwarding being overlapped with another node’s “burst.”



Takeaways:

- A complete Transceiver module

Meeting #5 - Package as Library

Goals:

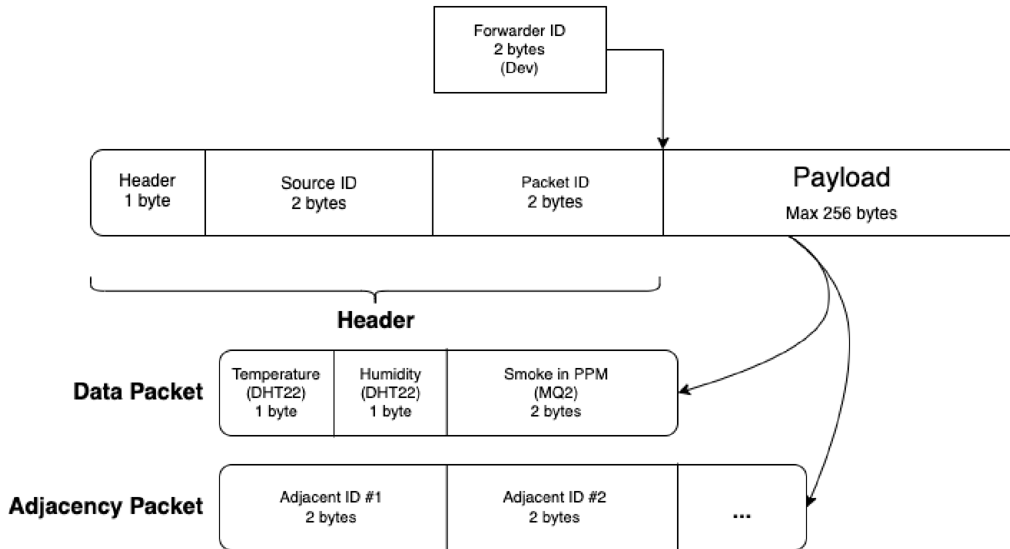
- Package code as Library
- Implement adjacency packets
- Create gateway code

Meeting Notes:

With the proof-of-concept done, the next step was to package and format the code neatly and as a library. This revamping as a library is necessary to reduce duplicate code, since the gateway program will be its own separate file. This revamping incorporated some abstraction into the program, bundling related code together and using structures to manage packets.

The packet structure is as follows: A header byte specifying the "type" of packet it is (i.e. how to interpret the payload), two bytes for the source ID, and two bytes for the packet ID. This header byte will be used by the server later when processing the packets.

The remainder of the packet is considered the "payload," its format customizable and user-defined. This abstraction allows for maximum flexibility and extension, making it a genuine "platform" or "framework" for future works to build on.



The two “types” of packets we currently have are *data* packets and *adjacency* packets. The first’s payload contains the sensor outputs, the latter being a list of IDs of the nodes it has received from – effectively, its “adjacencies.” This adjacency information can later be used to construct a graph of the nodes, allowing the operator to visually see the connections.

Takeaways:

- Library Model & Code

Meeting #6 - Implementing Gateways

Goals:

- Implement gateways

Meeting Notes:

The implementation of the gateway was quick, with little difference between the programming of the ESP-8266 and the Arduino Nano. It functions similar to the sensor nodes, except that it forwards via HTTP to the server instead of broadcasting via LoRa. For ease of transmission, the packet is sent in Base64 encoding in the query string. To test the HTTP networking abilities of the gateway, a dummy server was quickly scripted to log all received packets.

During testing, we found that after about 200 packets or so, the nodes would crash. We assumed it was a memory leak, and began logging the processor's remaining memory capacity at each step to identify the issue. As we had thought, the crash was caused by a memory leak from a pointer that was not deallocated.

Takeaways:

- Complete Gateway code
- Resolved critical issues

Meeting #7 - Implementing the API

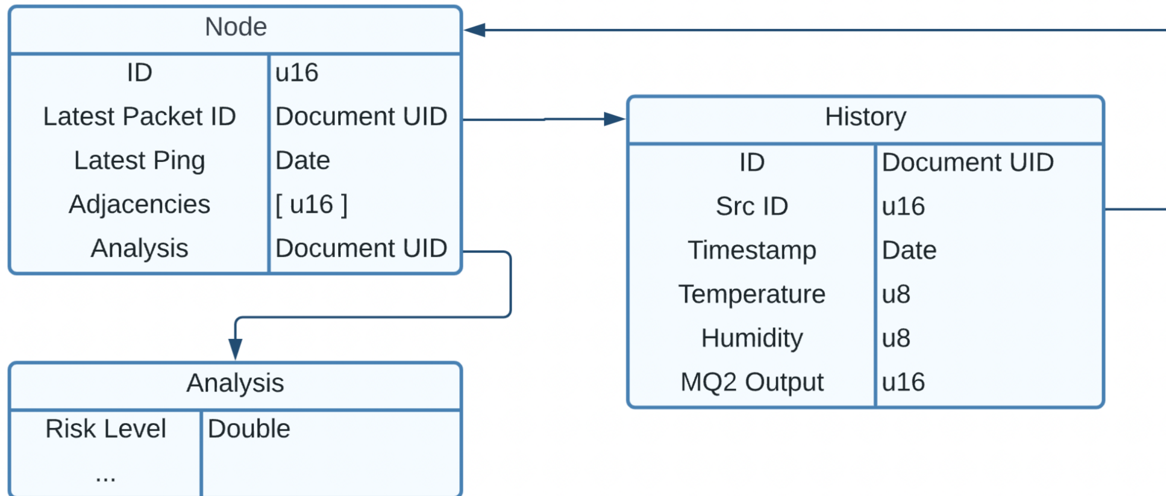
Goals:

- Create database schemas for nodes, gateways, and history entries
- Implement the get, post, put, and delete routes

Meeting Notes:

Following the structure that was set up for the API, we created schemas for nodes. Additionally, we created a schema to represent entries in the history table, which will be used for an interactive timeline and graph in the future. To do this, we kept track of the source node's ID, the packet ID, and the 3 attributes: temperature, humidity, and smoke level.

```
const mongoose = require('mongoose');

var Schema = mongoose.Schema;
var Types = mongoose.Types;

const HistorySchema = new Schema({
    srcID: Number,
    packetID: Number,
    temp: Number,
    humidity: Number,
    smokeLevel: Number,
},
    {
        timestamps: true,
        versionKey: false
    },
);

module.exports = mongoose.model('History', HistorySchema)
```



Above is a diagram of all the Schemas used in the project. We also implemented controllers to handle the incoming http requests, send the response back to the caller, and handle errors. We also utilized input validation to make sure that the API doesn't crash when a faulty GET request is made. To do this, we learned about all the HTML status codes and their purposes; for example, 200 corresponds to a successful request, 404 corresponds to a page not found, and 400 corresponds to a client error.

Takeaways:

- Complete Gateway code
- Resolved critical issues

Meeting #8 - Adding the Controllers

Goals:

- Abstract the nodes and gateways and adjust the API

Meeting Notes:

After further development of the project, we realized that to integrate the hardware and the backend, we needed to abstract the nodes and the gateways. Instead of having three schemas: Node, Gateway, and History, we decided to abstract the Nodes and Gateways because of repeated code. Additionally, the controllers were very similar for both the Nodes and the Gateways, so abstracting improved maintainability and readability.

Then, to further refactor the code, we edited the routes to only include the ones necessary to the frontend. While this makes the API not a true RESTful API, only the frontend was going to fetch the data, making extra routes unnecessary.

```
const express = require('express');
const router = express.Router();

const nodeController = require('../controllers/node');
const historyController = require('../controllers/history');
const netController = require('../controllers/net');

/* ## Details on each route can be found in the function headers in
their controller files ## */

/* ================ LoRaNet Routes ================== */
/* To be used by the gateways for data posting and configuration
tools for setup */

router.post("/packet", netController.packet_handler);
```

```
// PUTs & DELETEs nodes
router.put("/node", nodeController.put);
router.delete("/node", nodeController.delete);

/* ================ Client Routes ================== */
/* To be used by web client to fetch data */

router.get("/nodes", nodeController.get_nodes);             //
Returns all nodes, with their adjacency list
router.get("/gateways", nodeController.get_gateways);       //
Returns all gateways, with their adjacency list
router.get("/history/", historyController.get_all);         //
Returns an object where 'obj[id] = history'. A cutoff time can be
queried
router.get("/history/:id", historyController.get_by_node);  //
Returns the history of a node given a time interval


module.exports = router;
```

Takeaways:

- Finalized routes, controllers, and models

- Improved readability and maintainability

Meeting #9 - Sensor Integration

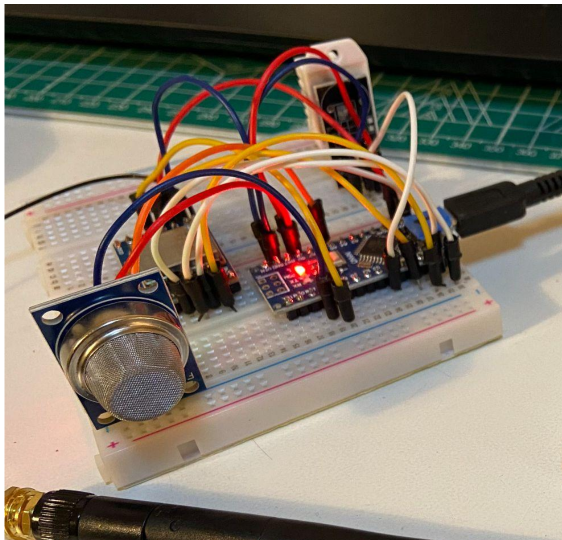
Goals:

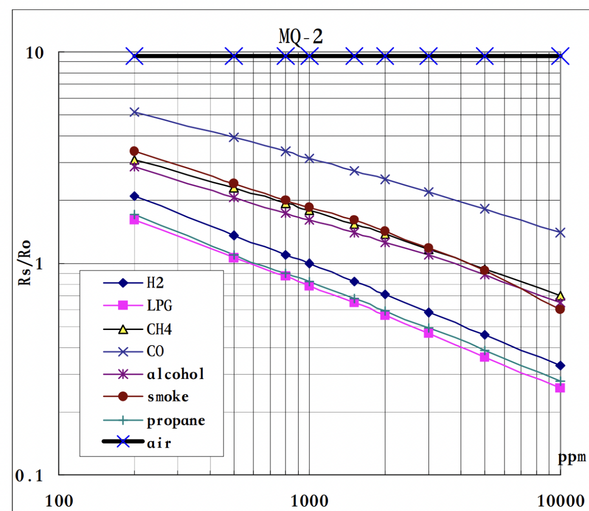
- Integrate Sensors into library & program
- Write necessary drivers

Meeting Notes:

With the network software architecture written, the next step is to integrate sensors into the framework. The sensors selected are the DHT22 temperature and humidity sensors, and the MQ-2 gas sensor. The framework, as mentioned above, is built around a customizable "packet" structure, making the integration process simple.

The integration of the DHT22 was rather trivial, as there already exists an Arduino library to operate the sensor. The wiring was also rather simple, the sensor having an analog output along with vin & ground. The software did not require much editing beyond inserting another field for the sensor data in the packet.





The MQ-2, on the other hand, required more work as it lacked a pre-made driver. Writing the driver ourselves required looking through the documentation and

datasheets to understand the interpretation of the output. The MQ-2 detects combustible gasses (LPG, H2, CO, Alcohol, Smoke, Propane) in the air through a tin-oxide wire that changes in internal resistance. The MQ-2 has two outputs, an analog and digital one. The digital output is a simple thresholder tuned by an onboard potentiometer. The analog output returns a variable voltage, from which we can derive its internal resistance. The numeric output can be derived from the fraction of its resistance over its resistance in clean air, which is logarithmically proportional to the parts-per-million output. By this, it is necessary that a calibration step be implemented in the driver to identify the normal resistance. With the driver packaged as a library, the integration was simple, much like the DHT-22's.

Takeaways:

- Integrated Sensors
- Complete MQ2 Driver

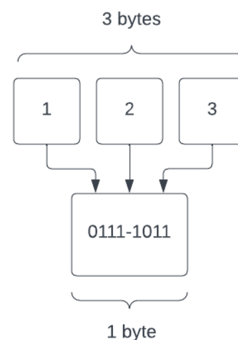
Meeting #10 - Optimization via Bits

Goals:

- Optimize package size via bit representation
- Implement bit-based parsers

Meeting Notes:

The LoRa module has advantages in range and power usage, however its one drawback is in its bandwidth. This calls for a minimal data footprint in our programs, thus our current legible string packet format had to go. The data footprint can be minimized to make use of every bit possible by representing numbers in their bit format as opposed to a human-interpretable string format. This significantly reduced the bits used; a three-digit number that would've taken 24 bits now only uses 8.



This involved re-writing the library's encoding & parsing functions, as well as helper functions to smoothen the process of manipulating raw bits in the packets.

One problem we ran into while writing this optimization was that the input buffer was treated as a "null-terminated string," meaning the module will assume a "0" byte to be the end of the message. Though the solution was rather simple – by using LoRa.write() instead of LoRa.print()" – it took us some time to identify.

Takeaways:

- Successfully Minimized Packet Size

Meeting #11 - AES Encryption

Goals:

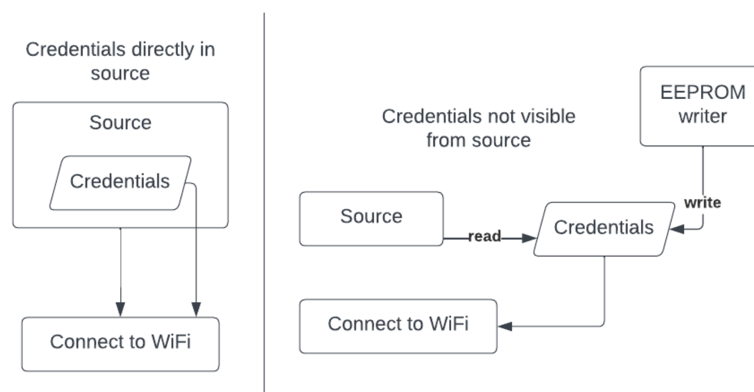
- Implement a simple wrapping cipher system

Meeting Notes:

Our next step was implementing some simple encryption; this development is more so to design a place for a security module in the framework and less so to implement a production-grade security protocol. It functions by enforcing that the message be encrypted / decrypted on parsing and encoding.

The encryption key is to be kept in EEPROM – read-only memory whose contents does not leave the program runtime. A dummy program was written to write data to be stored in EEPROM, and an initializing function was defined to extract the bytes out of the ROM.

This store-by-EEPROM strategy was also used to store the Wi-Fi credentials for the gateways. This is again to keep the credentials out of the source code.



A potential production-grade security protocol can be achieved by encrypting the payload section of the packet and leaving the header as plaintext. The cipher used

will be a public-private key algorithm, by encrypting using the server's public key, it ensures that the payload will only be parsed by the server. This does not compromise the network protocol, as the nodes maintain the ability to parse the header and identify the sources and packet IDs.

Takeaways:

- Completed AES Cipher System
- Completed EEPROM System

Meeting #12 - Server and Database

Goals:

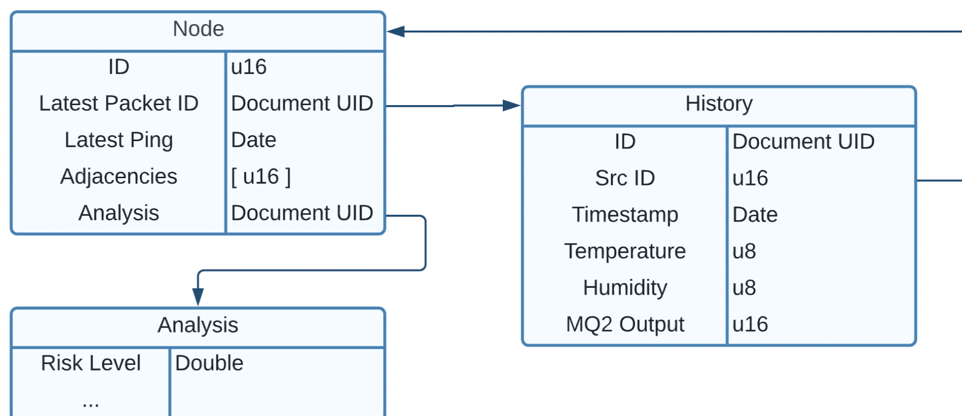
- Tweak server code for integration with database

Meeting Notes:

With the IoT software now done, we can finally integrate it with a real server and database. This was a matter of pulling up the server code that we developed back in the start of the project, and tweaking it to accommodate for some design changes.

The tweaking began by identifying the necessary routes for both the IoT network and the user interface, and trimming the rest.

One key change from our original database schema was the abstraction of gateways & nodes. While implementing, we saw a notable overlap in code between nodes & gateways, and opted to abstract the two with a boolean field making the distinction between gateways & nodes.



The next step was to re-write the server-side parser for the network's packets. For ease of transmission, the gateway packets are sent encoded in base64, which can be decoded easily. What followed was a simple switch-case program to process the

packets according to the header byte, writing to the database accordingly. The different “types” of packets are explained in a previous entry.

Takeaways:

- Complete database & API draft

Meeting #13 - Generation of Testing Data

Goals:

- Generate realistic clusters of node and gateway sensor data by studying trends of wildfire data in the past.
- Centralized generated node locations in an ideal arrangement around a given latitude and longitude
- Create a timeline of values for sensor data of individual nodes by studying trends of wildfire data in the past.

Meeting Notes:

For testing purposes, it is necessary that we have sample data to use. A configurable testing data generation program was written for this purpose, eliminating the otherwise manual process. This program not only needed to be configurable, but also generate a scenario that reflects a realistic scenario.

The node positions were generated to be within a configurable circular perimeter. The gateways were generated to be near said perimeter, as that would be the realistic layout of the network – gateways on the edge, nodes deployed deeper into the region.

Sample sensor data also had to be generated; this would be used to test the heat mapping interface and the graph view. This was done by studying the datasets available and arbitrarily selecting plausible ranges of values from it; this range is then coded into the generation program.

A key technique used in this program is bulk asynchronous scripting. Since this program involves writing a lot to the database, it is largely an I/O-bound program. To improve the performance – the speed of execution specifically – each writing to the database was executed concurrently as a list of short anonymous functions.

Takeaways:

- Complete sample data generator

Meeting #14 - Interface Development and Testing

Goals:

- Test network topology

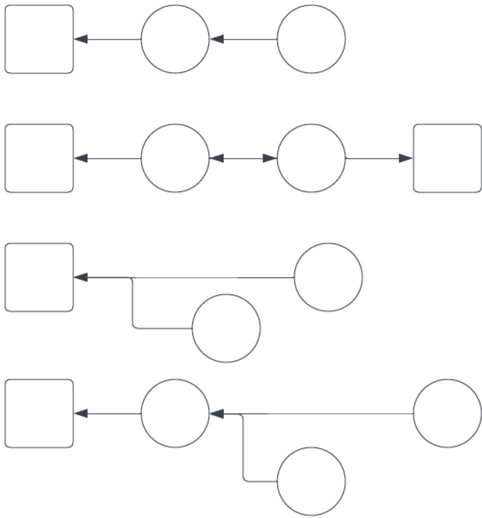
Meeting Notes:

While the others were developing the interface, the hardware lead was tasked with testing the network program. The most critical test was the topology, ensuring that the protocol is, indeed, capable of forwarding and generating a mesh network.

For this test, a new feature had to be implemented. In order to simulate a mesh topology, the nodes have to be able to identify and block packets from a certain node, emulating a disconnection. This was done by adding another field in the header to specify the address of which the packet was last forwarded from.

With the testing program written, testing went under way. The figure below lists the tested topologies.

A system was designed to emulate a set topology on the network. This was done by artificially creating a "disconnection" between nodes by blacklisting certain IDs. These fixed IDs and blacklists were written to EEPROM so as to not change the source code for every node. A special development field was added in the header to specify the "forwarder" – the ID of the node that relayed the message. This would be the field that the blacklist will be checked with to emulate discontinuities.



Takeaways:

- Constructed System for rapid testing

- Ensured functionality under different topologies

Meeting #15 - Client Interface Draft

Goals:

- Design schema for frontend
- Identify key incorporations
- Determine frameworks needed for development

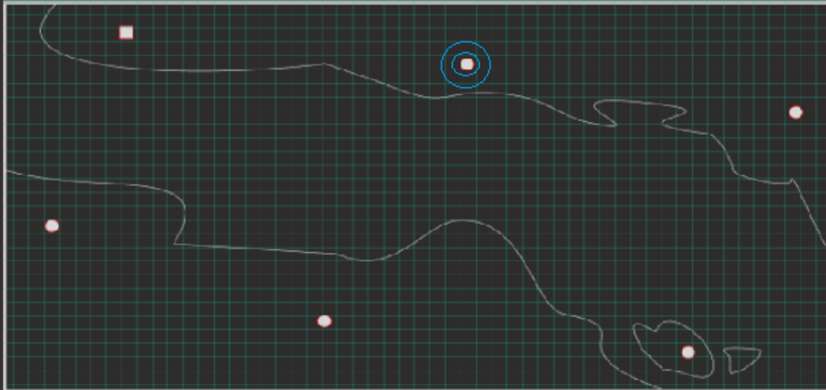
Meeting Notes:

Attempts for the beginning of the development of the client started today. We began by drafting general ideas for an intuitive user experience, with the commercialized aspirations of the projects demanded the incorporation of the following main aspects of our model:

- All nodes represented in a graphical manner imposed on top of real map data, with connections between nodes representing order of transmissions
- Visual difference betweens nodes and gateways to allow the user to differentiate between communications
- Clean and easy to interpret forms of the data at the current point in time
- Intuitive access to historical data and progression of sensor values over span of history table's data
- Access to risk values and interpretations of likelihood for development of fire in accordance to communication pathway

Based on these criteria, we went through several drafts and settled on the following layout for the main page of the client interface.

Node Name + Other Identification Labels Here (X,Y, etc.)



- Current Time
- Online (T/F)
- Humid Sensor Stats {

}
- Smoke Sensor Stats {

}

(Or if gateway, displays list of nodes and whether or not is online.)

the following would be the interactive timeline for the following node. you can click at any point on the timeline (how would we manage updating the scope of the timeline?), perhaps to get raw data. (we can have some sort of graphical interpretation of data progression attached to it if need be, overlayed over the timeline.)

Alerts would be visualized as a flashing red page which would alert the user which nodes were in trouble, their sensor values, and the severity of the alert.

Takeaways:

- Client Schema and Page Layout

Meeting #16 - Integration of Google Maps with React

Goals:

- Understand and interpret Google Maps API
- Be able to center the map at any coordinate

Meeting Notes:

After laying out the frontend, we decided to modularize each of its components using the React Component class, which allows reusable and customizable templates for aspects of the frontend - in this case, we wanted to use it for the map.

We ran through several different ideas for how we intended on integrating real visual data with the visual markers of the nodes and gateways, and ended up on the Google Maps API.

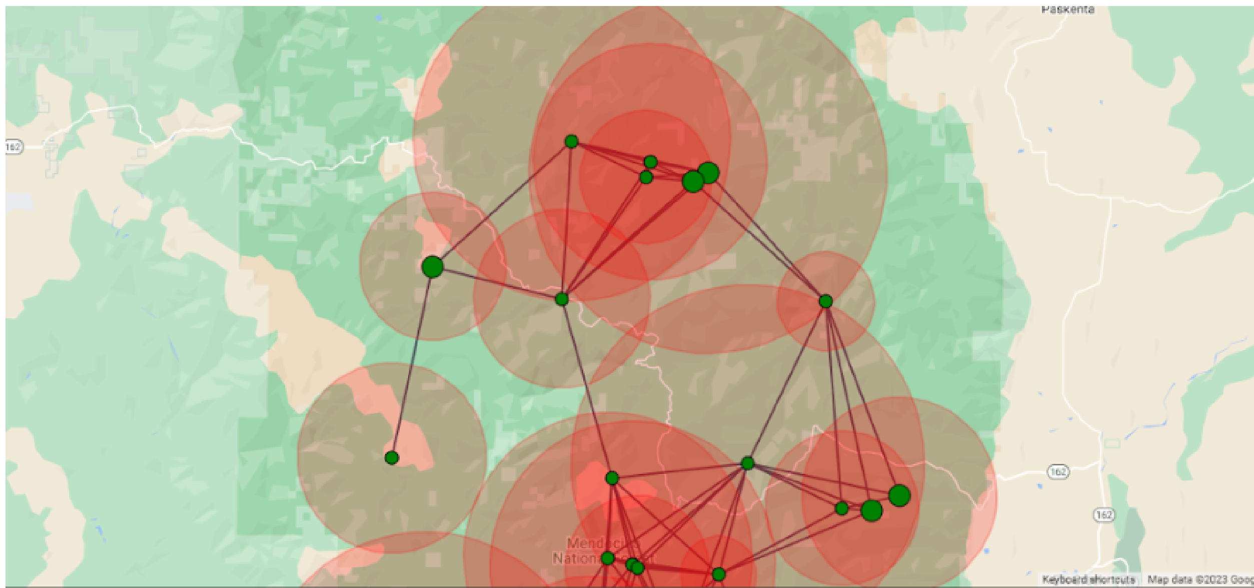
- Used Google Map Markers with given longitude and latitude data from nodes and gateways.
    - Larger circles were gateways, smaller circles were nodes.
- Each marker would have an onClick event that would fetch the data for that specific node and display it in the side tab.
- If a node or gateway was offline, the color of the marker would turn gray, otherwise it should be green.
- If a node were in risk, the icon would turn red.

Once the markers were completed, we decided to integrate the communication pathways through each nodes' adjacency list. We visualized those pathways using the API's Polyline component, which allowed us to dynamically render the line's

properties (i.e. start, end, line width, key points to curve, etc.) to avoid visual collision with other communications pathways.

With this completed, we also wanted a way to showcase the risk levels of each node at the last ping. To do this, we displayed concentric red circles of varying densities to showcase a “range” in which the risk would be most important to accommodate for. The higher the risk, the larger and more opaque the coloring of the circle range is.

The following is a diagram of the described interface in its current state:



Takeaways

- Dynamically constructable map from simulated node + gateway data

Meeting #16 - Handling Larger Client Interface

Goals:

- Transmit info from Map component back to Home page to display individual node or gateway data.

Meeting Notes:

With the Map component fully rendered, it was time to connect the isolated graphic to the rest of the main client page. In order to send information regarding which marker was being clicked to the main page, a callback function that communicated between child and parent page was employed. This allowed us to receive raw data, but we still needed to format said data into a more presentable and vibrant layout.

First, as per the schema, we allocated the top half of the screen for the map and the information display of the currently clicked node. Now, with the right side of the top half of the screen being occupied with the map, the left side was designated for info display. Utilizing a mix of JSX and native JS syntax alongside live fetches, we would be able to update data flow and recall in real time while still allowing dynamic rendering.

In order to retrieve the latest information, we took advantage of the timestamp element in the schema and retrieved the latest history entry of that given node ID.

The information displayed is described in the code snippet below.

```
<View style={styles.leftSquare}>
```

```
    <div>
        <div style={(calcOffline(retrieved.lastPing)) ?
styles.offlineTitle : ((retrieved.analysis.riskLvl > threshold) ?
styles.riskTitle : styles.onlineTitle)}>
            <h2>{("Node-") + retrieved._id }</h2>
        </div>

        <div style={styles.subInfo}>
            <ul>
                <li>{"Location: (" +
(retrieved.location.longitude).toFixed(2) + ", " +
(retrieved.location.latitude).toFixed(2) + ")"}</li>
                <li>{"Temp: " + (latestNodeInfo.temp).toFixed(2) +
"˚F"}</li>
                <li>{"Humidity: " +
(latestNodeInfo.humidity).toFixed(2)}</li>
                <li>{"Smoke Level: " +
(latestNodeInfo.smokeLevel).toFixed(2)}</li>
                <li>{"Last Updated At: " + moment(new
Date(retrieved.lastPing)).format('MMMM Do YYYY, h:mm:ss a')}</li>
                <li>{"Risk Level: " +
(retrieved.analysis.riskLvl).toFixed(3)}</li>
            </ul>
         </div>
    </div>
</View>
```

Takeaways:

- Dynamically loaded info display, constructing using queries from node and history tables

Meeting #17 - Graphically visualizing History Data

Goals:

- Visually appealing and interactive graphical interface to display historical sensor data of selected node/gateway.

Meeting Notes:

With the current information of the node's sensor data being displayed, we needed a way for the user to track the history of the node sensor data so that they could manually understand the progression over time.

Upon clicking the node, we sent a get request to retrieve all history table entries between a set interval of a week for that specific node. This would then be split into three different graphical figures. In order to switch between the three, we decided to use a dropdown that would allow us to toggle between the options. Upon clicking one, it would dynamically render the according graph over chronologically sorted data.

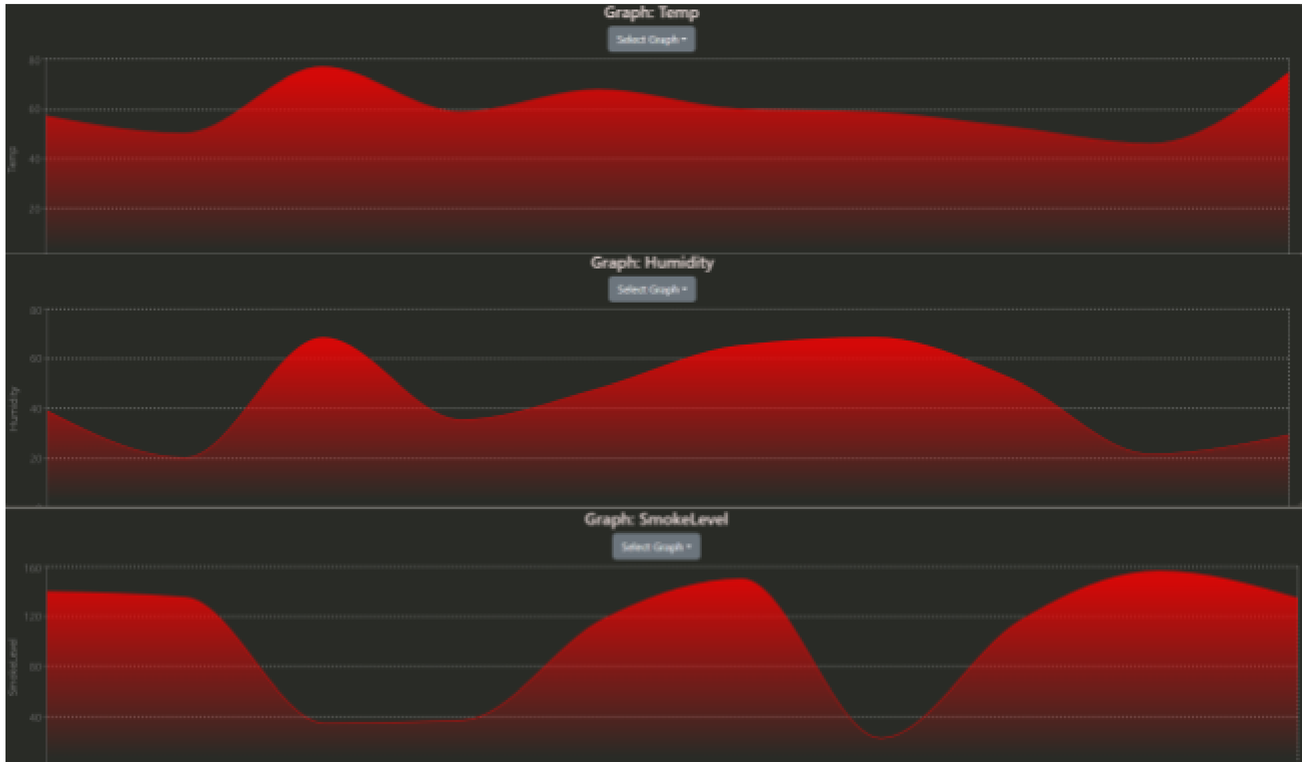
We represented the graphs using a react library known as Recharts, adding custom styling, color gradients, pop ups, and animations on top of it to beautify the user experience.

The data was also chronologically sorted prior to graphing.

```
import React, { useState, useEffect } from 'react';
import {AreaChart, CartesianGrid, XAxis, YAxis, Tooltip, ResponsiveContainer,
Area} from 'recharts';
import moment from 'moment';

function Graph(props) {
```

```
    const [data, setData] = useState([]);
    const [node, setNode] = useState();

    const PORT = props.port;

    useEffect(() => {
        fetch(`http://localhost:${PORT}/api/history/${props.nodeID}`)
            .then(res => res.json())
            .then(resJson => setData(resJson.results))
    }, [props.nodeID])

    useEffect(() => {
        fetch(`http://localhost:${PORT}/api/history`)
            .then(res => res.json())
            .then(resJson => setNode(resJson[props.nodeID]))
    }, [props.nodeID])

    const customSort = (a, b) => {return new Date(a.time).getTime() - new
Date(b.time).getTime()}
    data.sort(customSort);


    const CustomTooltip = (value) => {
        const idx = value["label"];
        console.log(idx);
        if (data[idx] === undefined) {
            return null;
        } else {
            return (
                <div style={{color: "#e6d7d5", backgroundColor: "#45433e",
margin: "5%"}}>
                    <ul>
                        <li>{"Temp: " + (data[idx].temp).toFixed(2)}</li>
                        <li>{"Humidity: " +
(data[idx].humidity).toFixed(2)}</li>
                        <li>{"Smoke Level: " +
(data[idx].smokeLevel).toFixed(2)}</li>
                        <li>{"Time Stamp: " + moment(new
Date(data[idx].time)).format('MMMM Do YYYY, h:mm:ss a')}</li>
                    </ul>
                </div>
            );
        }
    };
```

```
    if (node != null) {
        console.log("RISK: ", props.risk);
        const color = props.offline ? "#808080": ((props.risk == true) ?
"#ff0000" : "#32a852");
        if (props.value != undefined) {
            return (
                <div style={{margin: "auto 0"}}>
                    <ResponsiveContainer width="100%" aspect={5}>
                        <AreaChart data={data} margin={{left: 5, bottom: 50,
top: 10, right: 20}}>
                            <defs>
                                <linearGradient id="color" x1="0" x2="0" y1="0"
y2="1">
                                    <stop offset="5%" stopColor={color}
stopOpacity={0.8} />
                                    <stop offset="95%" stopColor={color}
stopOpacity={0} />
                                </linearGradient>
                            </defs>
                            <CartesianGrid stroke="#ccc" strokeDasharray="3,
3"/>
                            <XAxis tick={false} label={{ value: 'Timestamp' }}
interval={0} />
                            <YAxis dataKey={props.value} label={{ value:
props.value[0].toUpperCase() + props.value.substring(1), angle: '-90', dx:
-20}}/>
                            <Tooltip content={<CustomTooltip />} cursor={{
fill: "transparent" }} />
                            <Area type='monotone' dataKey={props.value}
stroke={color} fillOpacity={1} fill="url(#color)" />
                        </AreaChart>
                    </ResponsiveContainer>
                </div>
            )
        }
    }
}

export default Graph;
```



*Showcases all three graphs for a specific node - red coloring because this node has a high risk level

Takeaways

- Dynamic, animated graphical interface that loads history data for each node
- Completed client interface

Meeting #18 - Hardware Packaging and Analyzer

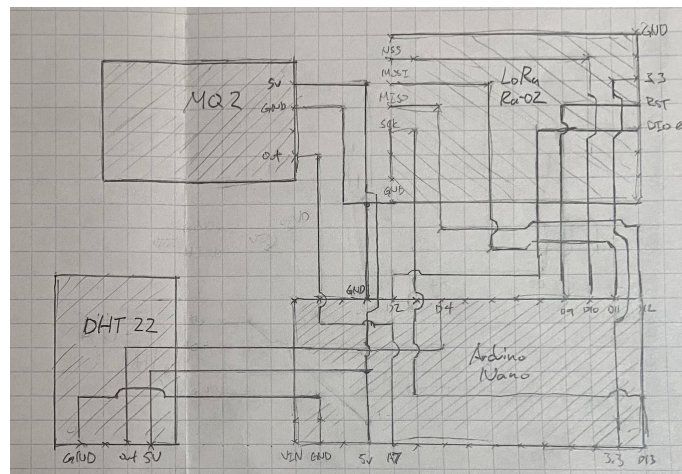
Goals:

- Design hardware schema
- Solder board
- Complete Analyzer module

Meeting Notes:

While work on the interface was continuing, development began on integrating an analysis module into the framework. We first implemented a dummy analyzer, a simple threshold for the analysis engine to ensure that the module is properly integrated. This would later be built on with stronger and more accurate models of analysis.

Another key step was to package the hardware into a small form-factor. This began by drawing up the circuit schematic for soldering. We settled on splitting the node from the battery, as we do not have a set battery package in mind. The soldering process took some time, but we were able to end the day with a complete circuit board. What is next would be the casing, which we are thinking of 3D printing.



Takeaways:

- Complete Abstract
- Complete circuit board
- Hardware case concept

Meeting #19 - Interface and Hardware Package

Goals:

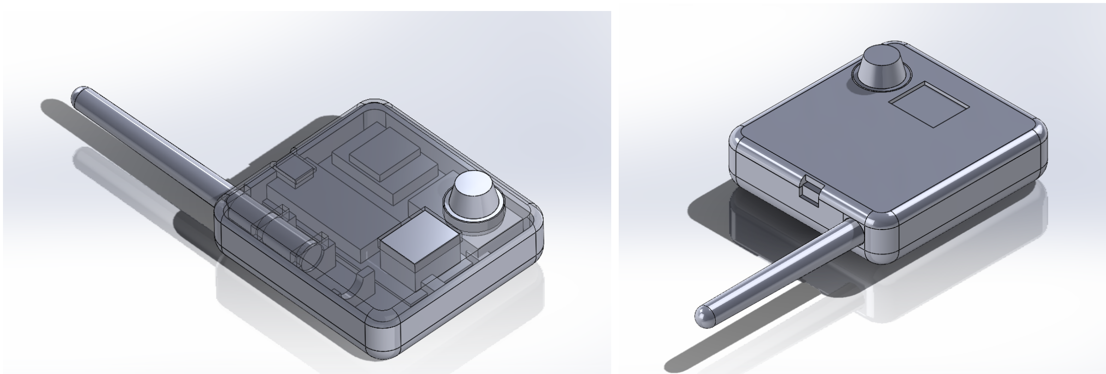
- Implement & Tidy up all features of the interface
- Complete Hardware Package

Meeting Notes:

The hardware casing's exterior was a simple box, though some details in the interior were needed to mount its components. The design process was simply sketching out the parts by hand, then transferring to a digital CAD model, before finally printing it.

There were also a few features we haven't yet got to implement on the interface; this includes visually rendering the adjacency graph between nodes. This idea was mentioned previously, but to reiterate, it involves drawing a line between two nodes that have received packets from each other, representing a connection. This will allow for a visual representation of the network and can help in the maintenance of the network.

We also spent some time tidying up the user interface, giving it a sleek and modern look, appealing to the eye.



Takeaways:

- Complete Hardware Package

- Complete Interface

Meeting #20 - Asynchronous Web Scraping

Goals:

- Scrape mass amounts of past wildfire data, quickly and efficiently

Meeting Notes:

We decided on using the FPA_FOD_20170508.sqlite wildfire dataset, which had essential data on wildfires spanning from 1992 to 2015. Additionally, this dataset contained information on the latitude and longitude of the fire, the starting and ending date, as well as less useful information on the names and causes of the fire.

To extract the data to be analyzed, we used the sqlite3 command line tool to parse through the sqlite file and turn it into a csv file, and then we utilized Python's pandas library, which we used to remove redundant data, and size down the dataset for easier web scraping by sampling random rows.

For the web scraping, the dataset only had information on the location and time of the fire, but we needed the temperature, humidity, and smoke values, as were provided by the hardware. After looking through 5 different weather APIs online, we settled on open-metio, a free API with no call restrictions, which also provided historical data.

After finding the API, the first idea we had was to use multiprocessing - spawning a multitude of processes, starting them, and joining them at the end. However, after extensive testing, we found that Python's multiprocessing module had an issue in which the processes could not join without hanging. Thus, we started looking into different options. Threading wasn't an option, because of slow runtimes, so we decided on using Python's builtin asyncio module to make asynchronous requests.

```python
def fetch_data_async(data: pd.DataFrame, d: dict) -> None:
    loop = asyncio.get_event_loop()
    future = asyncio.ensure_future(__fetch_data_all(data, d))
    loop.run_until_complete(future)


async def __fetch_data_all(data: pd.DataFrame, d: dict) -> None:
    tasks = []

    async with ClientSession() as session:
        for i in range(len(data)):
            task = asyncio.ensure_future(__get_data(data, d, i, session))
            tasks.append(task)

        _ = await asyncio.gather(*tasks)


async def __scrape_history(query: dict, session: ClientSession) -> str:
    queries = urlencode(query, safe=',')
    async with session.get(f'{__BASE_URL_HISTORY}?{queries}') as res:
        html = await res.read()

    return html
```

By taking advantage of coroutines, we managed to speed up the scraping by more than 500x with just a couple of revisions.

Takeaways:

- Format data and produce a correlation matrix

- Train a model for wildfire detection, or research existing models used in meteorology.

Meeting #21 - Correlation Matrix

Goals:

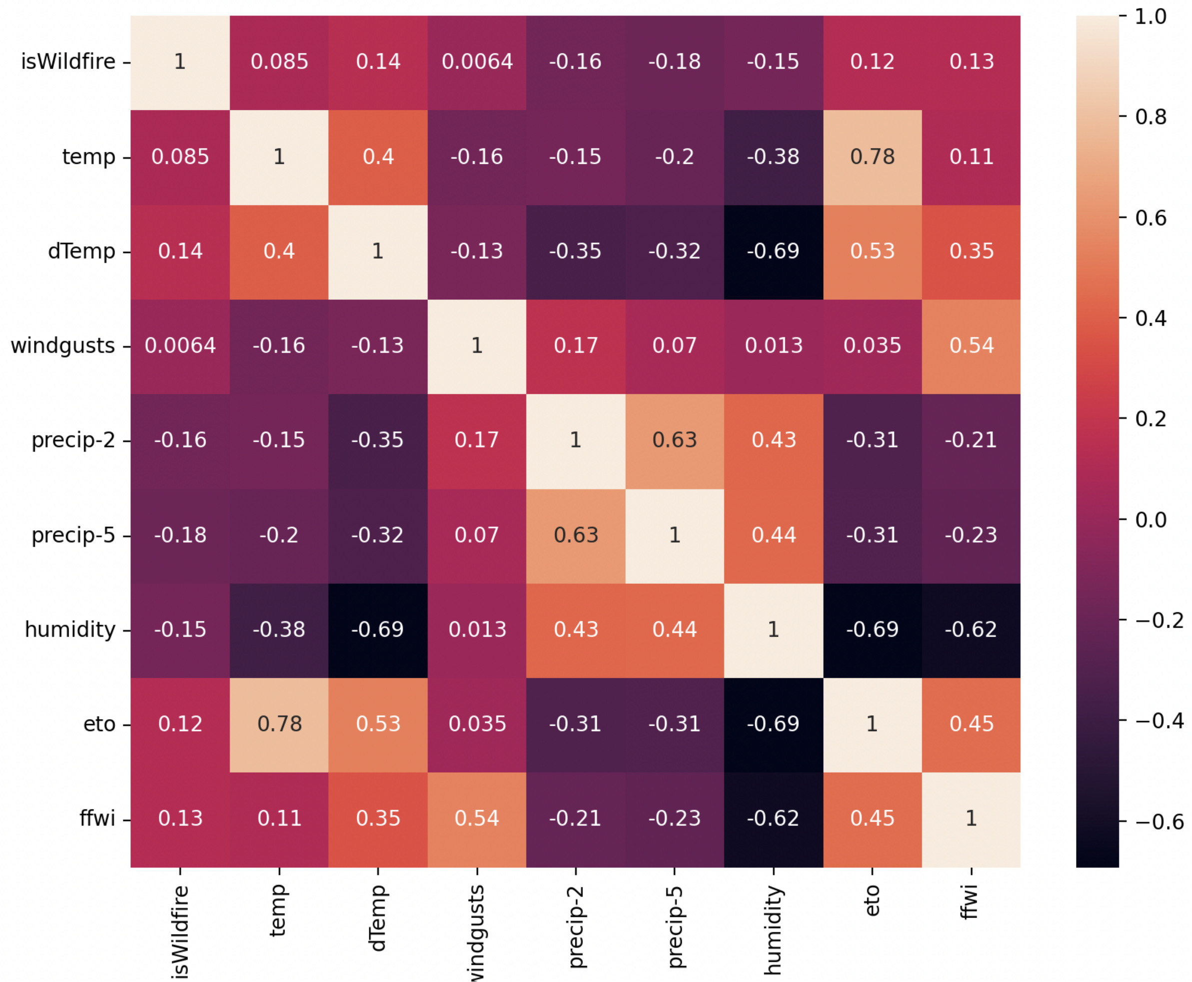
- Create a correlation matrix to identify trends in the data.

Meeting Notes:

To get data from both wildfires and non-wildfires, we designed the following algorithm:

1. Sample 10000 random rows from the wildfire database, and scrape the relevant data for each.
2. For each randomly sampled row, scrape the data from a year after the wildfire. Since the probability of two wildfires happening in the exact same place a year apart is extremely low, this makes it a good candidate for a non-wildfire data point. The pieces of data we decided to scrape were:
    a. Maximum temperature
    b. Minimum temperature
    c. Relative humidity
    d. Maximum wind gust speed
    e. Evapotranspiration rate - the amount of water lost in a certain time period from a cropped piece of land
    f. Fosberg Fire Weather Index - a measure of the risk of a wildfire given the temperature, humidity, and wind speed in a given location.
3. Use the resulting data to plot a correlation matrix, and identify which factors affect the risk of a wildfire more.

Surprisingly, there was no one factor that stood out as being directly correlated with the risk of a wildfire.



The factors that were somewhat positively correlated were dTemp (the difference between the highest and lowest temperatures), the Fosberg Fire Weather Index, and the evapotranspiration rate. Also, the factors negatively correlated with the risk of a wildfire were precipitation (5-day sum) and the relative humidity.

Takeaways:

- At a first glance, no factors are strongly correlated with the risk of a wildfire; however, this could not be indicative of an actual relationship between independent and dependent variables.
- Train a predictive model to output a risk between 0 and 1 given a set of data.

Meeting #22 - Support Vector Regression Model

Goals:

- Understand the difference between various regression models, specifically regular logistic regression, SVR, and Random Forest Classifier models.

Meeting Notes:

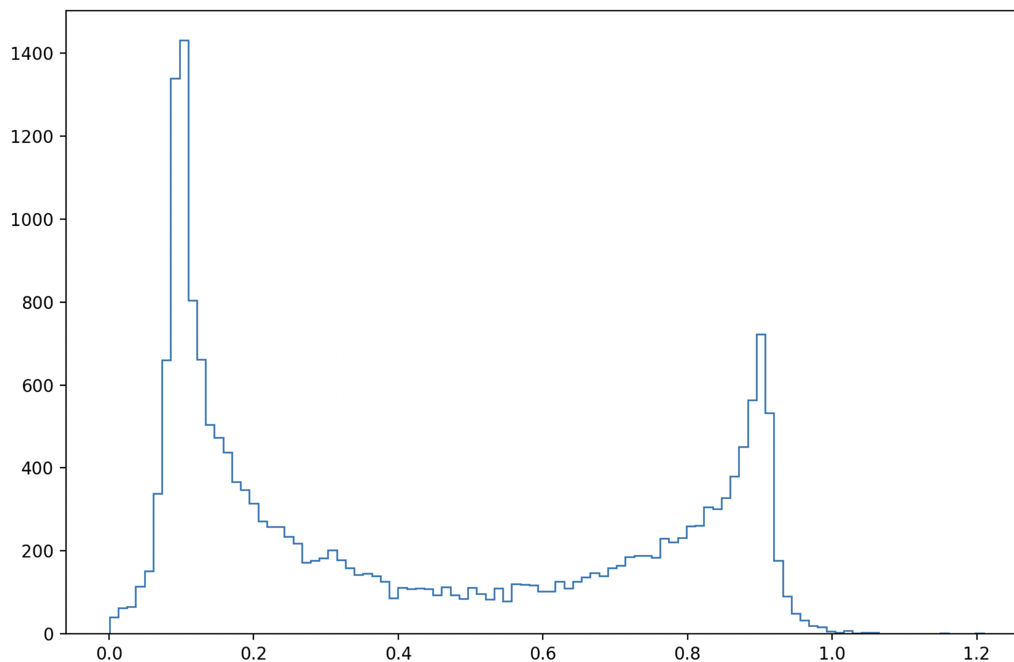
To begin, we imported the necessary libraries. In this case, the library scikit-learn, commonly used for training models, was utilized. Then, we split the data up into training and testing sets in preparation for the model.

We chose the Support Vector Regression Model because it worked well with limited data, and trained relatively quickly. Since we didn't have much data that we could quickly web scrape, this model proved the most beneficial to us. Additionally, the model is robust to outliers, which fits our use case perfectly because wildfires would have been caused by accidents not entirely dependent on weather events. Finally, the model performs relatively well compared to other regression models, which is ultimately why we chose it.

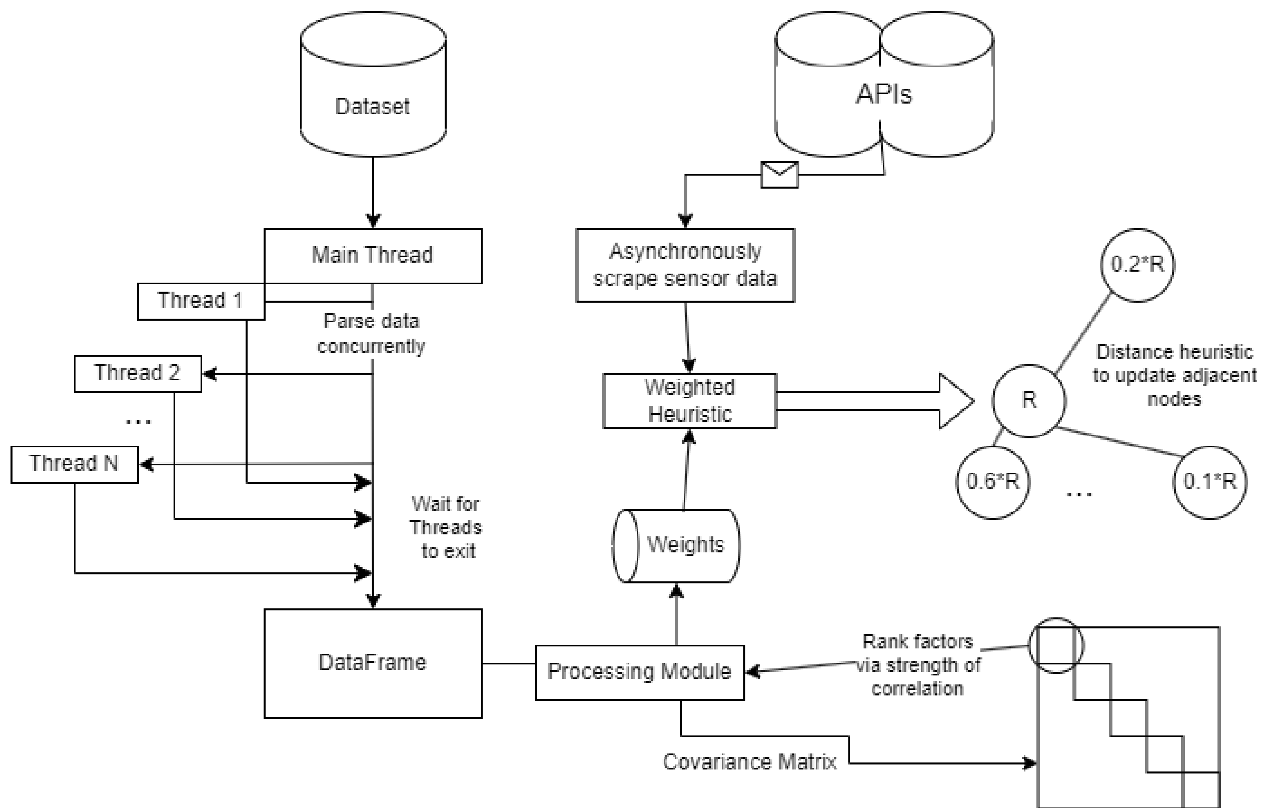
```
import pandas as pd
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error

data = pd.read_csv('./analyze.csv', index_col=0)
data = data.drop(['temp', 'windgusts', 'precip-2'], axis=1)

X_train, X_test, y_train, y_test =
train_test_split(data.drop('isWildfire', axis=1), data['isWildfire'],
test_size=0.2)

svr_model = svm.SVR()
```

```
svr_model.fit(X_train, y_train)
svr_pred = svr_model.predict(X_test)

print(mean_squared_error(y_test, svr_pred))  # 0.29299
print(mean_absolute_error(y_test, svr_pred)) # 0.43308
```

From sklearn's mean_absolute_error metric, we can see that the model most often overestimates or underestimates the risk by ~40%. Indeed, looking at a graph of the absolute error over 10000 data points, we can see that the majority of them are around 10% within the correct risk value, and 50% are within 40% of the correct risk value. In the future, by tuning the hyperparameters of the model, such as the regression degree, the regularization parameter, or heuristic, we can achieve a better performing model.



Then, to put everything together, we asynchronously scraped the data from the web, analyzed the relationships between the variables, and then trained a model on the data.



Takeaways:

- Export the sklearn model to JavaScript, then use a thresholding algorithm to determine whether or not to alert the user.

Meeting #23 - Dynamically Reloading Components

Goals:

- Have the frontend consistently poll the API for new changes, and reflect those changes on the map.
- Fix the map so it doesn't recenter

Meeting Notes:

We realized pretty quickly that react-google-maps had a bug which disallowed users from recentering the map without it not rendering whatsoever. Thus, we realized that the solution was to subclass React.Component, which allowed rendering every set interval, and then making the regular map Marker a part of that component. That way, changes to both the nodes and gateways could be reflected on the map, without interacting with it in any way.

```
class NodeMarker extends React.Component {
 constructor(item, index, props) {
   super();
   this.state = {
     curTime: null
   }

   this.timeIntervalLockout = 180;
   this.item = item;
   this.marker = new Marker({
     key: index,
     position: {
       lat: item.location.latitude,
       lng: item.location.longitude
     },
     icon: this.declareIcon(item),
     onClick: (() => {
       props.parentCallback(item);
     })
   })
 }
```

```
 calcOffline(timeStamp) {
   const currDate = new Date(moment().toISOString());
   const timeDate = new Date(timeStamp);
   //console.log(currDate, timeDate);
   let minutes = (currDate - timeDate) / (1000 * 60);
   return minutes > this.timeIntervalLockout;
 }

 declareIcon(item) {
   console.log("POINT TO BE RENDERED", item.location.latitude,
item.location.longitude);

   if (item.gateway) {
     if (this.calcOffline(item.lastPing)) {
       return {
         anchor: google.maps.Point(item.location.latitude,
item.location.longitude),
         path: google.maps.SymbolPath.CIRCLE,
         fillColor: "red",
         fillOpacity: 2,
         strokeWeight: 1,
         rotation: 0,
         scale: 10
       }
     } else {
       return {
         anchor: google.maps.Point(item.location.latitude,
item.location.longitude),
         path: google.maps.SymbolPath.CIRCLE,
         fillColor: "green",
         fillOpacity: 2,
         strokeWeight: 1,
         rotation: 0,
         scale: 10
       }
     }
   }
   else {
     if (this.calcOffline(item.lastPing)) {
       return {
         anchor: google.maps.Point(item.location.latitude,
item.location.longitude),
```

```
          path: google.maps.SymbolPath.CIRCLE,
          fillColor: "red",
          fillOpacity: 2,
          strokeWeight: 1,
          rotation: 0,
          scale: 6
        };
      } else {
        return {
          anchor: google.maps.Point(item.location.latitude,
item.location.longitude),
          path: google.maps.SymbolPath.CIRCLE,
          fillColor: "green",
          fillOpacity: 2,
          strokeWeight: 1,
          rotation: 0,
          scale: 6
        };
      }
    }
  }

  componentDidMount() {
    this.interval = setInterval(() => {
      this.setState({
        curTime: new Date().toISOString()
      });
    })
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      // To be completed
    )
  }
}
```

Meeting #24 - Re-evaluating Decision Model & Integrating w/ Client

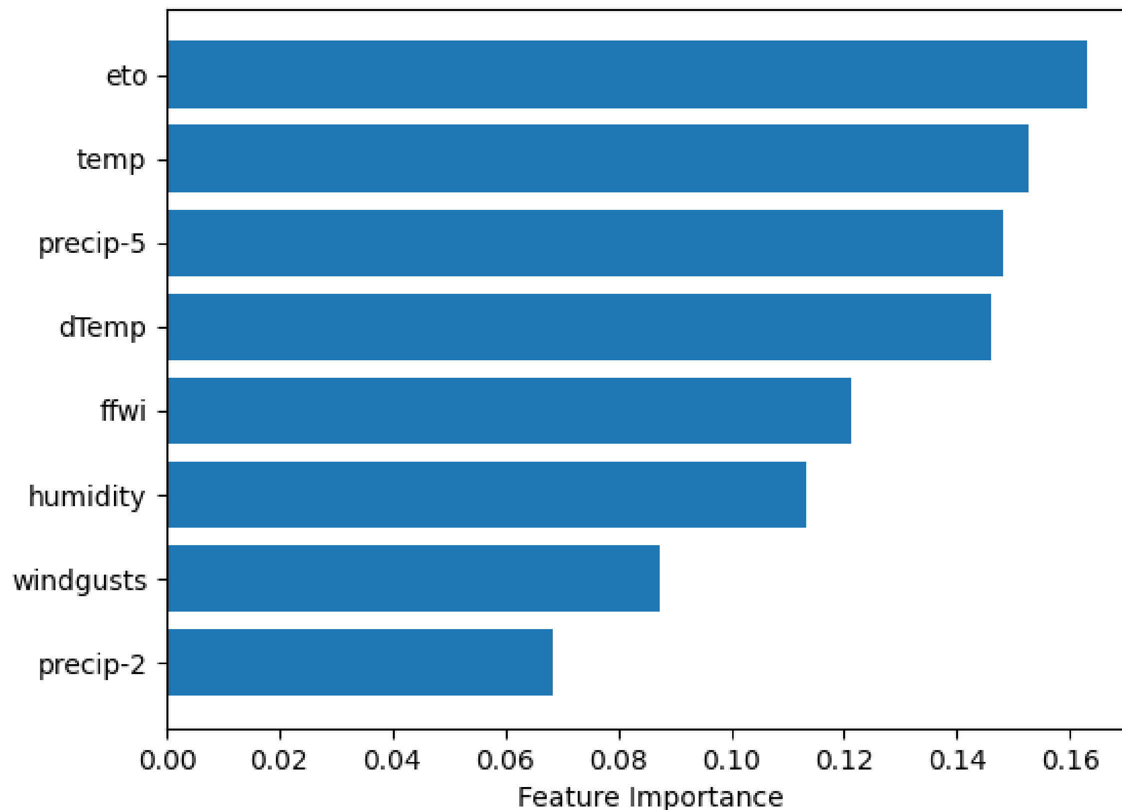
Goals:

- Include the model in the website, and evaluate the risks of a wildfire at a given node.
- Use a flood fill algorithm to propagate the risk value to other connected nodes.

Meeting Notes:

With the data scraped and available for analysis, we set to calculate the weights required to establish precedence for the thresholding heuristic. With our success rate with the SVM being relatively low, we chose to approach the decision making algorithm differently, rather associating a hierarchy of precedence for which thresholds are most important through a set of weights. These weights would be decided by an algorithm known as feature importance.

Utilizing a Random Forest Regressor, we can fit the model on the training data accumulated from earlier meetings, and receive a general estimate of the weights, as listed below:



The code to calculate these weights can be located below:

```
    import matplotlib.pyplot as plt
     data = pd.read_csv("analyze2.csv")
     print(data)
     y = data["isWildfire"]
     X = data.drop(columns=["isWildfire"])
     print(X)
     print(y)
     X_train, X_test, y_train, y_test = train_test_split(X, y,
 train_size=0.8, random_state=42)

     rf = RandomForestRegressor(n_estimators=150)
     rf.fit(X_train, y_train)

     sort = rf.feature_importances_.argsort()
     plt.barh(X.columns[sort], rf.feature_importances_[sort])
     plt.xlabel("Feature Importance")
```

```
plt.show()

print({k : v for (k, v) in zip(X.columns, rf.feature_importances_)})
```

Takeaways

- Successful calculation of weights for each aspect of the database using RFR and decision boundary

Final Takeaways/Future Plans

- Applicable power sources remain a challenge in all IoT projects, though developments in solar have the potential to address this issue.
    - This is a critical yet one of the most prominent issues obstructing IoT deployments; development in this area is most critical for the commercialization of IoT solutions.
    - Minimizing power consumption as a result becomes a major target for future research.
- Architectural, hardware, or software optimizations for minimal power draw
    - Following the previous point, optimizations on this front would contribute to the power supply issues
- Sensor optimization for set application.
    - The focus of the project being on the networking and the server-side framework, work on selecting optimal sensors and fields for analysis.
    - This would involve work on the environmental and meteorology fields.
- Automatic geolocation of nodes through trilateration of RSSI strength.
    - This is possible and has been done, however past works require that a single node has connections to at least 3 gateways.
    - Trilateration between nodes would require a thorough analysis on RSSI strength over distances and the environmental effects on this correlation.
    - An algorithm would also need to be designed for propagating coordinates between nodes.
- Security enhancements (key refreshes, one-way encryption, RSA, etc.)
    - As briefed in a previous log entry, there is potential in a RSA or one-way cipher for this networking protocol.
    - A concern could be in the computing prowess of the processors, limiting its ability to compute RSA quickly.
- Semi supervised pseudo-labeling analysis engine to compile data into a single risk measure

Thanks for reading!