Edge Computing For Smart Home Devices

***Abstract:* — A smart home system capable of functioning without connection to the internet or a central brain device. Modern smart home solutions largely have a central point of failure, being managed either by backend cloud server(s), by local "brain" devices, or a combination of the two. Should the cloud server, internet connection, or brain device go down, the system ceases to function. Our project aims to shift the interaction management and computing from the cloud server(s) and brain devices to the smart home devices themselves, so that the system can function without a connection to the internet.**

Index Terms:

- **Brain:** In modern IoT systems, the central processing point. Could be a backend cloud server or a dedicated device integrated into the smart home that functions as a central control.
- **Broker:** Interactions are managed using an abstract publish / subscribe message queue. This message queue is managed by a central service (broker) that manages subscriptions and passes published messages to subscribers.
- **Database:** Each node in the network will need to store information, such as sensor data, interactions, and user config. Database refers to the method by which each node will store this information locally.
- **Device:** A node in the network that focuses on actuation; i.e., it causes an observable physical change in the environment. E.g. a smart light that can turn on or off, a smart coffee machine that can begin to make coffee, etc.
- **Interaction:** An interaction is when data published by a sensor or shared by the system (such as the time) causes a device to perform an action. E.g. "If there is motion on my porch and it's dark out, turn on the lights".
- **Interaction Logic:** The collection of interactions defined by the user, how they are stored within the system, and how they are managed so that the interactions happen as intended.
- **Master:** The node(s) in the system that provides elevated functionality above other nodes, such as hosting the webapp or the broker.
- **Network:** The collection of nodes that interact together to deliver the functionality typically described as a "smart home"
- **Node:** Any piece of hardware in the smart home system that functions as part of the distributed system to deliver functionality to the user.
- **Sensor:** A node in the network that focuses on the production and distribution of data, but **not** on causing physical change in the environment. E.g. a motion detector that reads data from the sensor and notifies the rest of the network about the change.
- **Sensor Data:** Information collected and distributed by sensor nodes, intended to cause changes in the environment by device nodes. E.g. a reading from a motion detector or a light sensor.
- **System Config:** The collection of data necessary for nodes to interact with other nodes in the system. E.g. network id, identity of master node, nodes in the network and their local ip address, etc.
- **User:** The person who is interacting with the smart home system. E.g. a tech-savvy citizen who wants to automate their home without relying on the public cloud.
- **User Config:** The collection of data necessary to define a user's system. E.g. the interactions list.
- **Webapp:** The user-facing interface for monitoring and updating the smart home system.

I. Introduction

Since their introduction in the early 2000's, smart home devices have become increasingly popular. Devices such as the Amazon Alexa, the Ring Doorbell, and the Phillips Hue lightbulb have changed how many people interact with their homes. The management of these systems is largely done using some combination of the following:

1. Device sensor data is sent to a backend cloud server that keeps track of each device's state. Depending on user defined rules, the server may signal the devices in the home and cause them to perform an action.

2. Similar to the first method, sensor data is aggregated and sent to a central server. However this server is a device in the local smart home network, designated a “brain” device, that takes care of managing system interactions.
3. Some combination of the above two methods. Cloud server can function as a backup of the brain device and a way for the user to interact with the system.

This approach has a number of advantages. For example, it simplifies the system, as centralized solutions are generally easier to reason about and implement than distributed ones. It also becomes easier to add functionality to the system in the future by just updating the backend server, instead of requiring devices themselves to update. In addition, if any serious computation is required by the system (such as usage analytics), the backend can be scaled appropriately to provide the user a quick and seamless experience.

That being said, this approach has some serious disadvantages, perhaps the largest of which is the central point of failure. If the connection to the brain goes down, either by failure of the brain device or if the system loses connection to the internet, the smart home ceases to function properly. Devices may no longer interact correctly with each other, and in extreme cases users may no longer be able to control the devices in their own homes.

We aim to eliminate this disadvantage by shifting the device state storage and interaction management from the brain to the devices themselves. Our system aims to maintain the functionality of most modern smart home systems, namely:

1. Users should be able to interact with the system to both define interactions between their devices and view interactions that have occured in the past.
2. Devices should react as per their defined interactions in negligible time.
3. Devices can be controlled by the user through some sort of intuitive interface, such as a web or phone app.

The system should function normally even if the local network is disconnected from the internet, i.e., the above 3 requirements should function the same. However, if the network is down, then the user will be unable to monitor and interact with their devices outside of the local network.

II. Design Requirements

When we first designed the system around which this project is centered, we had planned to have physical hardware to test and demonstrate on. However, about halfway through the project the Covid-19 crisis forced us to transition to remote learning. This meant that our project needed to pivot away from using hardware, and so we were forced to transition our system to function in the cloud. While this is somewhat contrary to our stated project goal, we feel that by limiting the nodes to only interacting with other nodes in the network, we can successfully simulate that the system is "disconnected" from the internet (See section IV for a more detailed explanation of how this changed our project). Due to our pivot, we had to change or get rid of some of our requirements. These changes are reflected in the table below.

Requirement	Original Specification	Revised Specification	Justification
Sensor input to device action latency under normal use	<100 ms	< 10 ms	Took into account the lack of latency of the AWS network
New device commission time	< 10 minutes	Removed	We didn't have the time needed for this feature.
Past sensor data viewable	4 years	Removed	Didn't have a good way to test this and couldn't accurately simulate storage sizes on AWS
Cost per node	< $75	Removed	Cannot compare the cost of AWS compute to RPI compute
Internet resiliency	Device interactions should continue to function normally	Device interactions should continue to function normally	Could still easily prove that this is fulfilled
Downtime in case of master node failure	< 5 seconds	< 5 seconds	There was no reason to change this
Downtime in case of non-master node failure	None	None	There was no reason to change this

Our system's goal is to operate fully without needing a connection to the internet. Therefore, the above requirements are derived largely from existing smart home solutions. The most critical of these requirements are "sensor input to device action latency under normal use" and "internet resiliency". We believe these to be the most important requirements because the former is necessary in order to match current smart home solutions (at least in functionality), and the latter is required in order to solve the problem we have identified.

Here is how we tested the above and what we found:

- <u>Sensor input to device action latency under normal use:</u> We tested this using timestamps. After confirming that the different nodes' system times were in sync, we logged (with timestamps) each time a sensor value was published and each time a device acted on an interaction. This test was done under our "full system load"; i.e., 5

- nodes: 2 sensors, 2 devices, and 1 combined

- one at random. We then took the difference between the time that the sensor posted the update and the time that the device acted on it, and averaged it over 119 trials. This gave us an average of 2.76 ms, well under our requirement of 10ms.

- Internet resiliency: This was more difficult to test automatically. We tested this one by confirming (both in our design and in our code) that the nodes only made requests to other nodes in the network. If this system were to be ported onto physical devices, then as long as they are on the same local network, our system would work. This fulfills our internet resiliency requirement, as the system does not need access to the internet to work.

- Downtime in case of master node failure: We tested this requirement by forcing nodes to die, and then programmatically timing how long it took for the new node to become available. We did this by writing a script that did the following:

  - Note that given the state of the system, we know which node will become the new master after a failover. Let's call this node "node 2", and the current master node "node 1"

  - Curl the webapp at the IP address of node 1 in a loop until it fails.

node. We defined 6 interactions and then picked

Take a timestamp

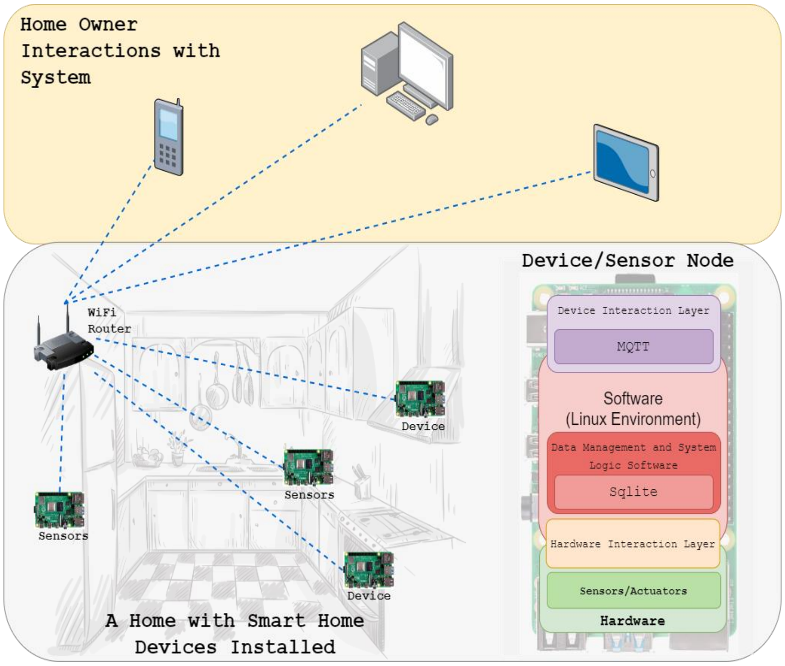
  - Curl the webapp at the IP address of node 2 in a loop until it succeeds. Take a timestamp

  - Print the difference between the two times.

Doing the above procedure and averaging our results over 20 trials, we found that on average, the system experiences 8.668 seconds of downtime. Unfortunately, this is greater than our requirement of 5 seconds. We believe that this is largely because the nodes are hosted on a virtual machine over which we don't have full control. As such, spinning up new applications is expensive, and could explain why we were unable to reach our 5 second goal.

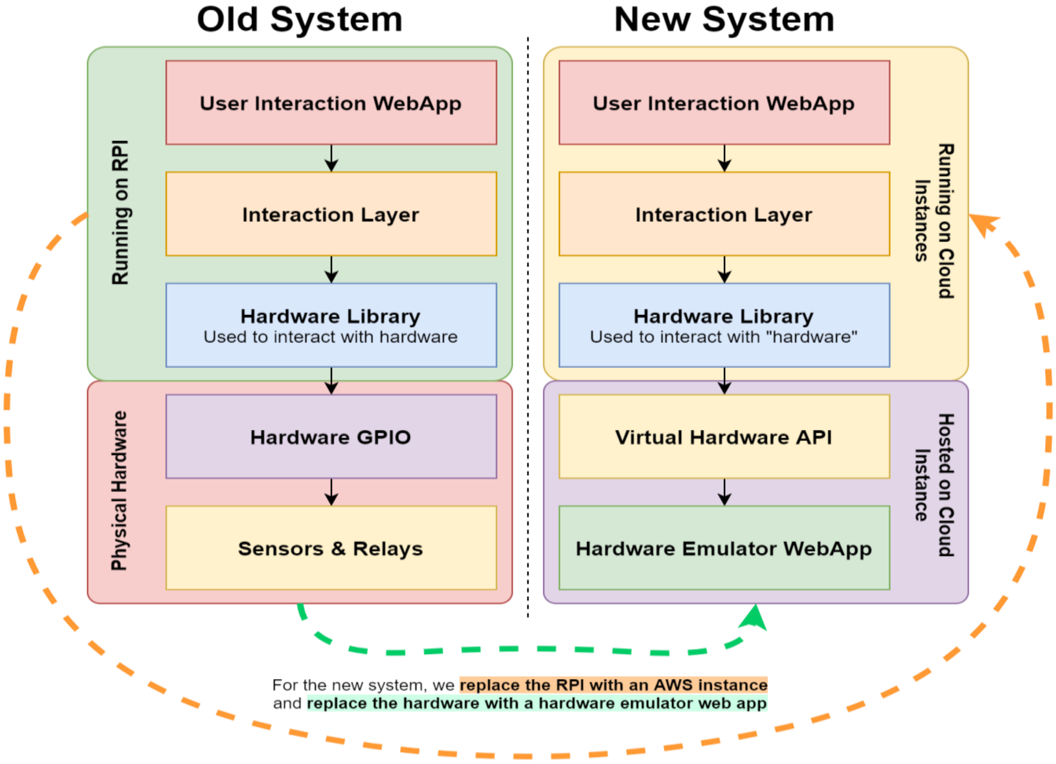
- Downtime in case of a non-master node failure: This was tested by randomly bringing down non-master nodes in the system. We confirmed through interaction logs that other devices continued functioning normally and logged the dead node's death. Moreover, any interactions that were defined that did not include the dead node continued to function as normal. This makes sense, since we designed the system to be compartmentalized so as to limit the effect of failures. This fulfills our requirement of having no system downtime in the event of a non-master node failure.

III. Architecture and/or Principle of Operation


Home Owner Interactions with System
WiFi Router
Device
Sensors
Sensors
Device
A Home with Smart Home Devices Installed
Device/Sensor Node
Device Interaction Layer
MQTT
Software (Linux Environment)
Data Management and System Logic Software
Sqlite
Hardware Interaction Layer
Sensors/Actuators
Hardware

As stated earlier, our original architecture (see right) was not possible for us to build after our transition to remote work. Originally, we had planned to use Raspberry Pis as the compute platforms for each of the devices, WiFi as the networking platform, and build custom PCBs on top of the Raspberry Pis. With this, we could run software for the Interaction Layer on the Raspberry Pis, communicate with other devices using MQTT over WiFI and store system data on the Raspberry Pis.

We needed to change this drastically for the project to continue remotely, since we were unable to test and develop hardware when we were all separated from one another. Because of this, we chose to transition our work to the cloud, and develop on a similarly powerful cloud instance to the Raspberry Pis that we had planned to use. It was important to us that what we were working on could at some point be transitioned to these Raspberry Pis. Providing a way of interacting with hardware while working with hardware was still a challenge for us however. We decided to emulate the hardware using a seperate web app also hosted in the cloud. We wanted to give the developer, us, the same ability to control the environment around the hardware and test the system without using actual hardware. The transition of the system and the final architecture of the system is outlined in the diagram below.



IV. Design Trade Studies

A. *Hardware Platform of System*

Limitations:

a. Hardware platform needs to be able to host MQTT broker as per interaction layer

b. Hardware needs to support a local database

Our system had two options in regards to which platform the smart devices should be built upon: Raspberry Pi's or NodeMCU's. While the ESP8266, the NodeMCU model we could use, would work well for our application, it is manufactured in China. Usually, this would not be a problem, but recent events concerning the Covid-19 Novel Coronavirus make the shipping time for this component unpredictable. As such we decided to go with the Raspberry Pi, specifically the model 4, as it is already in inventory.

When deciding on which router to use, we considered our 100 ms device interaction latency requirement. The way an interaction will occur within our system is as follows: sensor node polls hardware (software, hardware), sends updated data to broker (network latency), broker determines subscriber(s) to send the data to (software), the broker sends the data to the subscriber node (network latency), and the device node then acts on that data (software, hardware). The code required to implement the software portions of this interaction is of negligible size, and with the raspberry pi 4 processors will run in milliseconds. However, the network latency introduces a potential bottleneck. According to [3], typical wireless network latency is around 20 ms. Since we have 2 data transfers over the network (see section 5, system design), a typical homeowner would see 40 ms of latency. Our goal in selecting a router was to pick one that can at least satisfy this requirement, so as to replicate the typical user's network setup.

A typical TCP packet consists of 40 bytes, and an MQTT message has 9 bytes of overhead (excluding payload). We intend to allot space for a 64 byte payload, amounting to 113 bytes of data per transfer. To transfer 113 bytes of data twice within 100 ms, we need to transfer 226 bytes per 100 ms = 2260 bytes per second = 2.26 KBps = 18.08 Kbps. Since most modern routers function in Mbps connection speeds, not Kbps, any router we select should be able to meet the "20 ms latency" cited in [3].

We chose to use the TPlink C1200 router because it fits in our budget and is powerful enough to satisfy the latency requirements outlined above.

The previous design trade studies were for the original system however. In the final system, after our pivot, we faced these same design questions but couldn't solve them in the same way. We knew that the best only way to continue this project would be to host it in the cloud. We chose AWS because we were all at least vaguely familiar with the workflow and felt comfortable with it. We choose python as our development language for the same reasons, familiarity and comfort. We didn't feel like there was anything we needed to do in this system that couldn't be done with python.

For the Hardware Emulator, in place of the actual hardware, we chose Django and the Django REST Framework because we were familiar with it, and also because it is extremely fast and easy to develop with. This remained true throughout this project, since we never experienced an issue with Django that took longer than an hour to fix, and never had a feature that we wanted to add that took more than a day of work to figure out.

B. *Database Platform of System*

Our system will use a database to store information locally on each of the nodes. A lot of thought went into whether we should have a shared distributed database, a local database on each device, or a combination of the two. To begin this comparison, let us look at the information that needs to be stored in the system:

1. For each node in the network:
   a. Serial number
   b. IP address
   c. Whether it is the master node (hosting webapp and broker)
   d. Whether it is a sensor or a device

e. Whether it is up and running, and if not when it last was
f. Display name and description to display on the webapp

2. Defined interactions
   a. Trigger sensor
   b. Target device
   c. Condition
   d. Action
   e. Display name and description to display on the webapp
3. Sensor data
   a. Value
   b. Timestamp

Initially, we considered storing everything of the above in a shared distributed database. This would greatly simplify interactions, as all nodes could read the shared data, and state would become “centralized”. However, this introduced 2 major problems. The first problem is that often in distributed databases, you have master node(s) that can write to the database, and slave node(s) that function as replicas of the data. Since our system is intended to be used in home settings, the number of devices will be limited, and so there will be a small number of master nodes through which all devices must write. Given that our timing requirements require polling with frequency greater than 100 ms, this would necessitate a write over the network to the shared database once every 100 ms *per sensor*. Besides the fact that this unnecessarily clogs up the user’s network, it introduces a pointless bottleneck into the system. A second problem with this approach is that sensor data for each sensor would exist on all of the nodes. We felt this level of replication is unnecessary, and forces us to use more storage than is otherwise needed.

This motivated us to shift towards a combined approach. Of the above data, items 1 and 2 would be defined in a shared database, and item 3 would exist locally on each sensor node. The sensors could publish the data to the broker as it is produced, and store it locally. If historical data is ever needed, it can be requested from that sensor. At first glance, this approach seems to fix all the problems of the first one. However, we were unable to make a strong argument for the existence of a shared database storing items 1 and 2.

Our reasoning behind this is as follows: items 1 and 2 consist of a (relatively) small amount of data, and this data will not be written to frequently. These items will largely be changed only when a user logs into the web interface to change their configurations, which is unlikely to happen more than a few times a day. As such, it doesn’t really necessitate the overhead and complexity involved in using a distributed database. Instead, we decided it would be far simpler (and equally effective) to have each node store a local replica of items 1 and 2, and have sensor nodes also store item 3. Whenever items 1 and 2 are updated (through the master node), it will send an update to the other nodes, and they will update their local database’s version of the configuration.

Throughout this process, we considered a number of distributed and local databases, including Redis, Apache Cassandra, MongoDB, Hbase, and SQLite. However, our decision to shy away from a distributed database made our criteria shift a little bit. In terms of storage footprint, read speeds, and simplicity of use, SQLite won over the others. In addition, it could more easily run on the single board computers required by the rest of our system, as compared to some of the other options (such as Cassandra) that require the JVM.

When we pivoted to have the system in the cloud, we wanted to replicate as much of the functionality of the initial system design, rather than fully leverage all that is available on modern cloud hosting services. As such, we kept the database design choice the same after pivoting the project, so that the interaction layer could still be hosted on actual hardware (with minor modifications to the code).

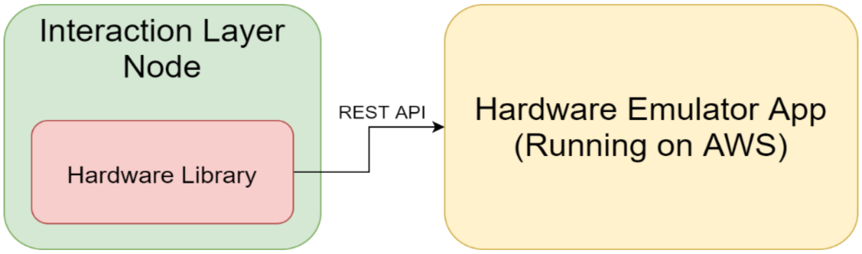
When choosing a backend web framework, we decided to go between Flask and ExpressJS. Both are simple and lightweight, relatively easy to set up and deploy. Ultimately, we chose to go with Flask because it would be easier to integrate with the interaction layer. Since the interaction layer is written in Python, if we went with ExpressJS like originally planned, we would have had to find a way to connect the interaction layer python library with a javascript function. Using Flask, on the other hand, entailed only importing the interaction layer library at the top of the backend script, thus making our lives much easier.

V. System Description

Logically, our system consists of three parts, the hardware layer, the interaction layer, and the UI web app. Having a system that we could easily split into three different distinct parts made splitting up work easy and helped us integrate with each other much simpler. This section delves into more detail about each of those sections of our system and discusses how these layers interact.

A. *Hardware Systems*

In the final iteration of the hardware layer, physical hardware was emulated using a webapp and a hardware library utilized by the interaction layer. The emulator app was made using Django, a common Python web framework, SQLite, and a REST API. The following diagram is an illustration of the components of the hardware layer and their interconnects.



1. The first part of this layer is the Emulator Web App. This was created using Django and the Django REST Framework to allow the library, the second part of this layer of the system, to communicate with the Emulator easily. Instead of creating a custom UI for developers to interact with the emulator, we decided to use the built in Django admin page because it already had a simple and easy to use way to display all this information. We stored all information about each device and sensor in the SQLite database connected to the emulator and used the REST framework to communicate with the library.
2. The library was created using Python and utilized the requests framework to communicate with the Emulator. The library was initialized and interacted with using a JSON string that the device instance gives to the library. The JSON string outlines all characteristics of the device, including sensors, pins connected to hardware, actuators, and value types, so that the Emulator and any future hardware can easily interface with this library. The different pieces of hardware (sensors and actuators) are stored in a library object for the interaction layer, so that if the device has multiple pieces of hardware it can access all of them using the library object.

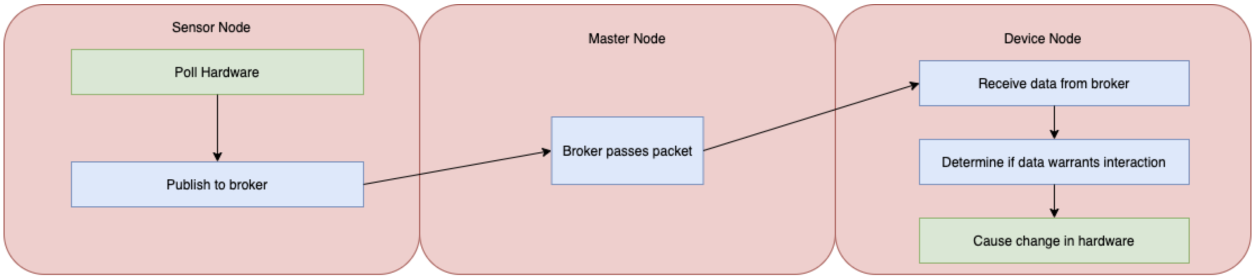
B. *Interaction Layer Systems*

a. <u>Device interactions:</u> Interactions are managed using MQTT. The master node in the system runs the broker, and all sensors and devices connect as clients. Sensors publish to sensor-specific MQTT topics, and devices subscribe to topics required by user-defined interactions
This interaction looks like this:
Note that the broker process may be running on any device or sensor, *including* potentially the device or sensor involved in the transaction. Sensors poll the hardware for updates every 50 ms, and if the value has changed from the previous poll (greater than a delta defined in the sensor spec as acceptable jitter), it publishes this information to an MQTT topic of the form "[device_serial_number]/data_stream". Devices with interactions tied to a sensor subscribe to that sensor's publishing topic; if multiple sensors are involved in a single interaction, the device can subscribe using wildcards. When the sensor publishes a changed value, the device will receive it through the broker, and if it matches what was defined in the interaction, it will cause a change in the hardware (such as a light turning on or off).
Data packets sent to and from the broker will be a json of the following form:

```
{
   'device_serial_number' : unsigned long,
```

‘data’ : primitive type (specific to the device doing the transmitting)}

b. Local device storage: Devices will store config and sensor data in a local SQLite database.



c. Hosting the webapp: See “master selection” below. One master node will exist in the system, and will host the webapp. Users can access the webapp to interact with the system.

d. Master Selection: The master will be selected by looking at the devices that are currently in the network and up (based off the heartbeats), and selecting the node with the numerically lowest serial number.

e. Config updates: Whenever the user changes configuration (i.e. adds a node to the network, defines a new interaction), all nodes in the network should be informed of this change so they can update their local copy of the config. This is implemented on top of our MQTT layer. All nodes subscribe to a topic named “config_updates”. Whenever there is a config change, the master node publishes to this channel, and the nodes update their local version of the config.

f. Device heartbeats: This will be implemented on top of our MQTT layer. Initially we were planning on implementing the heartbeats by having each node publish to heartbeats topic, and all nodes reacting to these updates. However, we realized that this added potentially unnecessary bandwidth use and complexity. We discovered two interesting MQTT features that allowed us to pivot this design decision in a different direction:

i. Topic will messages: when an MQTT client connects to the broker, it can specify a “will” message for a specific topic. If the client ever disconnects without going through the proper disconnect procedure with the broker, the broker assumes the node has died, and publishes that message to the will topic. What this means is that all nodes can subscribe to the “heartbeats” topic, and publish a will message to that topic. That way, the only time a heartbeat gets sent is when a node dies. This significantly reduces network bandwidth usage, and accomplishes the same goal with greater precision (nodes will know immediately when another node has died, as opposed to when they next expect a heartbeat message to be published).

ii. On_disconnect async callbacks: many client-side MQTT libraries provide async callback functionality, allowing the client code to register callback functions that are invoked when certain things happen. One such callback, named on_disonnect in the client library we chose to use, allows you to register a callback that is invoked if the connection with the broker is unintentionally broken. What this means is that if the master node ever dies, all other nodes will be immediately notified and can begin the master failover procedure locally.

C. *Webapp systems*

a. The frontend will be built using React. Light, fast, and easy to prototype with, React can also be containerized easily using Docker, making it extremely portable. There shall be a home page, used a dashboard to view other pages, a devices page to view devices in the system, a page for each device, an interactions page to view interactions in the system, and a page for each interaction. On the interactions page, there should also be a form to create new interactions.

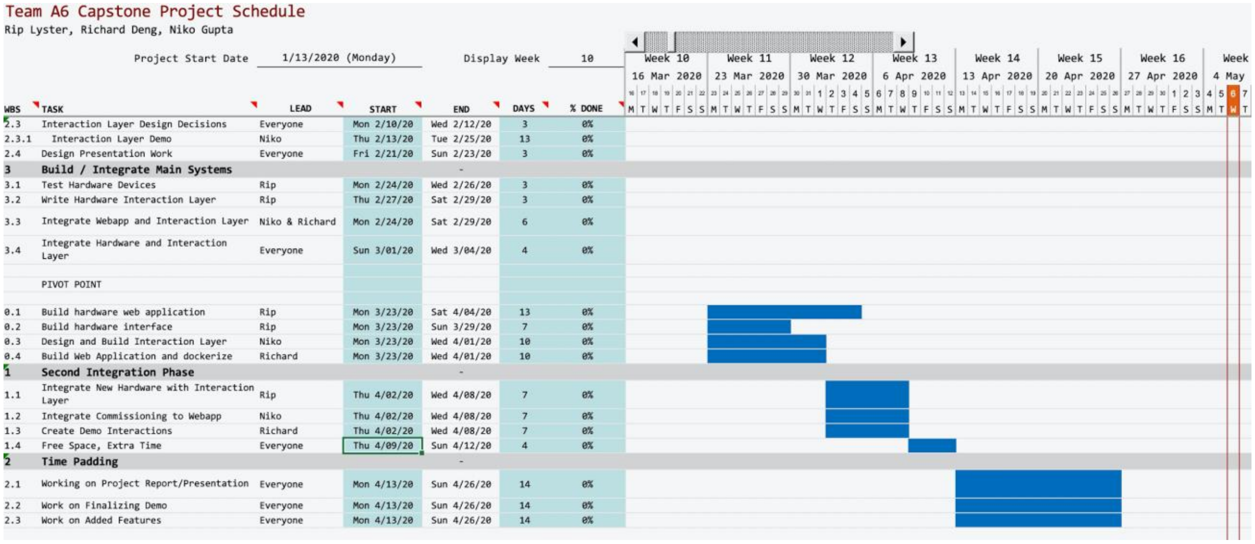
b. Backend-Express
   The backend of the web application will be built using Flask. Simple, lightweight, and minimal, Flask covers all of the systems backend requirements without anything extra. Models can be easily implemented using JSON, which makes integration with the rest of the system seamless. Since the interaction layer is written using Python, using a Python based backend seemed appropriate for the smoothest integration.

   i. Communicates to system via REST api

c. Mounted on Raspberry Pi using a Docker container

VI. Project Management: Change to new Gantt chart

A. Schedule



Team A6 Capstone Project Schedule

Rip Lyster, Richard Deng, Niko Gupta

Project Start Date 1/13/2020 (Monday) Display Week 10

WBS	TASK	LEAD	START	END	DAYS	% DONE
2.3	Interaction Layer Design Decisions	Everyone	Mon 2/10/20	Wed 2/12/20	3	0%
2.3.1	Interaction Layer Demo	Niko	Thu 2/13/20	Tue 2/25/20	13	0%
2.4	Design Presentation Work	Everyone	Fri 2/21/20	Sun 2/23/20	3	0%
3	**Build / Integrate Main Systems**			-		
3.1	Test Hardware Devices	Rip	Mon 2/24/20	Wed 2/26/20	3	0%
3.2	Write Hardware Interaction Layer	Rip	Thu 2/27/20	Sat 2/29/20	3	0%
3.3	Integrate Webapp and Interaction Layer	Niko & Richard	Mon 2/24/20	Sat 2/29/20	6	0%
3.4	Integrate Hardware and Interaction Layer	Everyone	Sun 3/01/20	Wed 3/04/20	4	0%
	PIVOT POINT					
0.1	Build hardware web application	Rip	Mon 3/23/20	Sat 4/04/20	13	0%
0.2	Build hardware interface	Rip	Mon 3/23/20	Sun 3/29/20	7	0%
0.3	Design and Build Interaction Layer	Niko	Mon 3/23/20	Wed 4/01/20	10	0%
0.4	Build Web Application and dockerize	Richard	Mon 3/23/20	Wed 4/01/20	10	0%
1	**Second Integration Phase**			-		
1.1	Integrate New Hardware with Interaction Layer	Rip	Thu 4/02/20	Wed 4/08/20	7	0%
1.2	Integrate Commissioning to Webapp	Niko	Thu 4/02/20	Wed 4/08/20	7	0%
1.3	Create Demo Interactions	Richard	Thu 4/02/20	Wed 4/08/20	7	0%
1.4	Free Space, Extra Time	Everyone	Thu 4/09/20	Sun 4/12/20	4	0%
2	**Time Padding**			-		
2.1	Working on Project Report/Presentation	Everyone	Mon 4/13/20	Sun 4/26/20	14	0%
2.2	Work on Finalizing Demo	Everyone	Mon 4/13/20	Sun 4/26/20	14	0%
2.3	Work on Added Features	Everyone	Mon 4/13/20	Sun 4/26/20	14	0%

B. Team Member Responsibilities

The breakdown of work aligned with our experience. Richard worked on the UI webapp, since he has a large amount of experience with web app development, Niko took charge of the interaction layer since he has experience with computer systems and operating systems, and Rip worked on the hardware and the hardware library since he has experience with embedded systems and hardware. The pivot tested all of our skills, since we had to work on everything in a completely new context.

C. Budget

Preliminary Budget

Name	Quantity	Unit Price	Total Price
Raspberry Pi 4 model B	6	55	330
TPLink C1200 Router	1	50	50
Mr. Coffee Coffee Maker	1	25	25

PIR Motion Sensor	3	2	6
DS18B20+	3	5	15
Mini USB Microphone	2	5	10

Final Budget

Name	Quantity	Unit Price ($)	Total Price ($)
AWS T2 Micro Instance	1230 (hrs)	0.0162	20

AWS T2 Micro Instance refers to the type of machine we rented from Amazon Web Services (AWS). Our system used many instances at the same time (sometimes up to 7 or 8), which is why there are 1230 hours logged. We count 1 hour as 1 instance running for 1 hour, so, if 7 nodes were running for 2 hours total, in this period we would have logged 2 * 7 = 14 hours.

D. Risk Management

In the Gantt chart, we scheduled two weeks for integration. After hearing many warnings from past groups, we know the danger of not allocating enough time to put together the pieces of our project.

In designing our demo system around a morning routine, we have narrowed the scope of our project. This tightens the focus of the group, lessening the chance that non-essential work is done, which is important given how long the project is active for.

E. *Sudden changes*

When we were all instructed to move home, all three of us were quite shocked. We were on spring break in Florida when we realized that we had to pivot our project. Since we were all together, planning how to pivot could be done in person. We analyzed the three big parts of the project, and figured out how each part would change.

The web application interface used to control the system didn’t change. Instead of the web application running on a device in the system, it would now run on an AWS server. This ultimately didn’t change the design of the web application at all. The web application development plan resumed with no further obstacles.

The interface layer needed to communicate with both the web application as well as the hardware layer. Since we no longer had physical devices, the interaction-hardware connection needed to be changed. Additionally, since the web application UI is hosted on an AWS server now, the interaction layer needs to communicate differently with the web app.

Finally, we pivoted from using real physical devices to using a hardware emulator. Instead of building the smart devices using raspberry pi’s, we built an api that simulates the hardware IO. If the interaction layer needed the value of a certain sensor, it would call the api with the device’s id. If the interaction layer wanted to change the value of a device (e.g. a lightbulb or a coffee pot), it would call a different endpoint on the api with the target id and target value. Pivoting to an api instead of real physical devices let us test the system remotely. If we didn’t pivot this part, we would have to all be at the same place to integrate or test the interaction layer and the hardware layer.

One particularly important reason for choosing to implement a fixed api to govern interactions between the interaction and hardware layers is that it improves portability. If we were to someday port this system onto physical devices, the hardware library’s api could remain the same, while the implementation could be rewritten. This would allow the interaction layer to continue functioning with minimal changes.

VII. Related Work

One interesting project done in edge computing was done by Tanmay Chakraborty and Soumya Kanti Datta in late 2017 [4]. They identified the biggest problem in the IoT space as fragmentation caused by having many competing IoT device producers all producing similar products using different communication protocols. This has lead to a world where it's difficult for users to get their devices to work together, which impedes the development of unified smart home systems.

The solution they proposed (and implemented) is an architectural prototype that "exploits the concepts of edge computing, virtual IoT devices, and the internet of things to create an interoperable home automation solution" [4]. While not directly the project that we worked on, it's interesting to see that others are looking into how to take advantage of edge computing to improve home IoT systems.

Another interesting project, "Vigilia", was done in late 2018 [5]. This project aimed instead at making smart home systems more secure. In particular, they focused on "shrinking the attack surface of smart home IoT systems by restricting the network access of devices" [5]. They did this by creating an open framework to limit network access only permissed devices.

Our project was also partially driven by privacy; we identified central storage of smart home data by a 3rd party as a potential security risk, and aimed to create a system where smart device data never leaves the closed home system. It's interesting to see how others have approached solving the security problems posed by the growing IoT industry.

VIII. Summary

Our system was able to meet most of our adjusted specifications. Our latency requirement was 10ms, but our slowest test over twenty trials was just under 3ms. The system also met the internet resiliency requirement. This specified that the system shall work even when not connected to the outside internet. We tested this by placing all nodes in the system in a private security group, thereby restricting any traffic from any source outside the system and isolating the network.

A requirement we failed to meet was the master failover timing requirement. We specified that once the master node fails, a new master shall rise within 5 seconds. After 20 trials, the mean failover recovery time was over 8 seconds: 3 seconds more than our requirement. After timing how long it takes an AWS T2 micro server to spin up, we realized that we might have set an unrealistic requirement, as it often takes more than 5 seconds to boot up an instance to a state the webapp can run on.

To improve our system if we had more time, we would transfer the system to real hardware instead of software emulators. Basically, if we had more time and could meet up in person, we would have implemented the hardware devices on Raspberry Pi's instead of AWS cloud instances. Everything else about the project would remain the same.

References:

[1] User study on device commissioning times (Note names anonymized for privacy)

Person	Device	Commission time (minutes)
1	Tp link smart bulb	3
1	Amazon Alexa	10
2	Amazon Alexa	8
3	Ring camera	5-10
3	Sonos smart speaker (wifi pairing)	5-10
3	Ecobee thermostat (same pairing process as sonos)	5-10
4	Amazon Alexa	10-15
5	Smart security Camera	10
6	Amazon Alexa	10
7	Smart Bulb	10

[2] https://ecfsapi.fcc.gov/file/6520222942.pdf

[3] Sui, Kaixin, et al. Characterizing and Improving WiFi Latency in Large-Scale Operational Networks. Characterizing and Improving WiFi Latency in Large-Scale Operational Networks. http://zmy.io/files/mobisys16-WiFiSeer.pdf

[4] T. Chakraborty and S. K. Datta, "Home automation using edge computing and Internet of Things," 2017 IEEE International Symposium on Consumer Electronics (ISCE), Kuala Lumpur, 2017, pp. 47-49, doi: 10.1109/ISCE.2017.8355544. https://ieeexplore.ieee.org/abstract/document/8355544

[5] R. Trimananda, A. Younis, B. Wang, B. Xu, B. Demsky and G. Xu, "Vigilia: Securing Smart Home Edge Computing," 2018 IEEE/ACM Symposium on Edge Computing (SEC), Seattle, WA, 2018, pp. 74-89, doi:

10.1109/SEC.2018.00013.
https://ieeexplore.ieee.org/document/8567658