Edge Computing For Smart Home Devices

Author: Nikolas Gupta: Electrical and Computer Engineering (ECE), Carnegie Mellon University (CMU); Ripley Lyster: ECE,CMU; Richard Deng: ECE, CMU

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 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 or private 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 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 as far as 4 years in the past.
2. Devices should react as per their defined interactions within 0.1 seconds.
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.

Requirement	Specification	Justification
Sensor input to device action latency under normal use	<100 ms	< 0.1s considered "instantaneous" by the FCC [2].
New device commission time	< 10 minutes	After conducting interviews of friends and family about their smart home device usage and commission times, we found that on average devices took around 10 minutes to set up from unboxing to use. More detailed results can be found at [1].
Past sensor data viewable	4 years	Typical device lifetime is 2-4 years. Given a typical sensor, all of its historical data should be viewable from the webapp.
Cost per node	< $75	Smart home devices typically < $75, some are a bit over. E.g Phillips Hue ($55), Alexa Echo ($50), Ring doorbell cam ($100).
Internet resiliency	Device interactions should continue to function normally	If the home router disconnects from the internet, the system should still function normally.
Downtime in case of master node failure	< 5 seconds	Based off user research we conducted, users were willing to wait 5 seconds for a webpage to load before they gave up. As such, we require that in the worst case (failure case), a new master node is elected and made functional in under 5 seconds.
Downtime in case of non-master node failure	None	The only functionality lost if a non-master node goes down should be the interactions that device is involved in. I.e. if a sensor goes down, any interaction relying on that sensor's input cannot work, as the sensor is not functional.

II. Design Requirements

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 intend to test the above to demonstrate that our system meets the requirements:

- Sensor input to device action latency under normal use: Using a high - speed camera (iPhone X can shoot at 240 fps, ~4 ms per frame) we will film a sensor being triggered and the resulting interaction. Assuming the system is running normally (master is up), this should be under 100 ms.
- New device commission time: We will make an extra device that is not connected to the network. Using a simple set of instructions, the average user should be able to pair it into the system in less than 10 minutes (we will test with at least 10 people).
- Past sensor data viewable: We do not intend to run our system for 4 years to demonstrate the storage capacity. Rather, we will provide detailed calculations of our data schema and frequency of "saving data", along with equations showing how much data can realistically be expected to be stored in 4 years. Our system should have a strictly larger amount of storage than that.
- Internet resiliency: We will purchase a simple router that we will use to simulate a user's home network. This router will be disconnected from the internet during our demo, and the device interactions should work as defined in our requirements.
- Downtime in case of master node failure: Given a set of a sensor and a device node, with an interaction defined between the two, the device should react to sensor input within 100 ms under normal operation. We will disconnect the master node and continuously provide input to the sensor, demonstrating that the device reacts after no more than 5 seconds.

Downtime in case of a non-master node failure: This can be tested similarly to the master node failure case. Given a set of a sensor A and a device node, and a separate sensor node B (that is not a master), given continuous input to A, the device should react even if B is disconnected.

III. Architecture and/or Principle of Operation

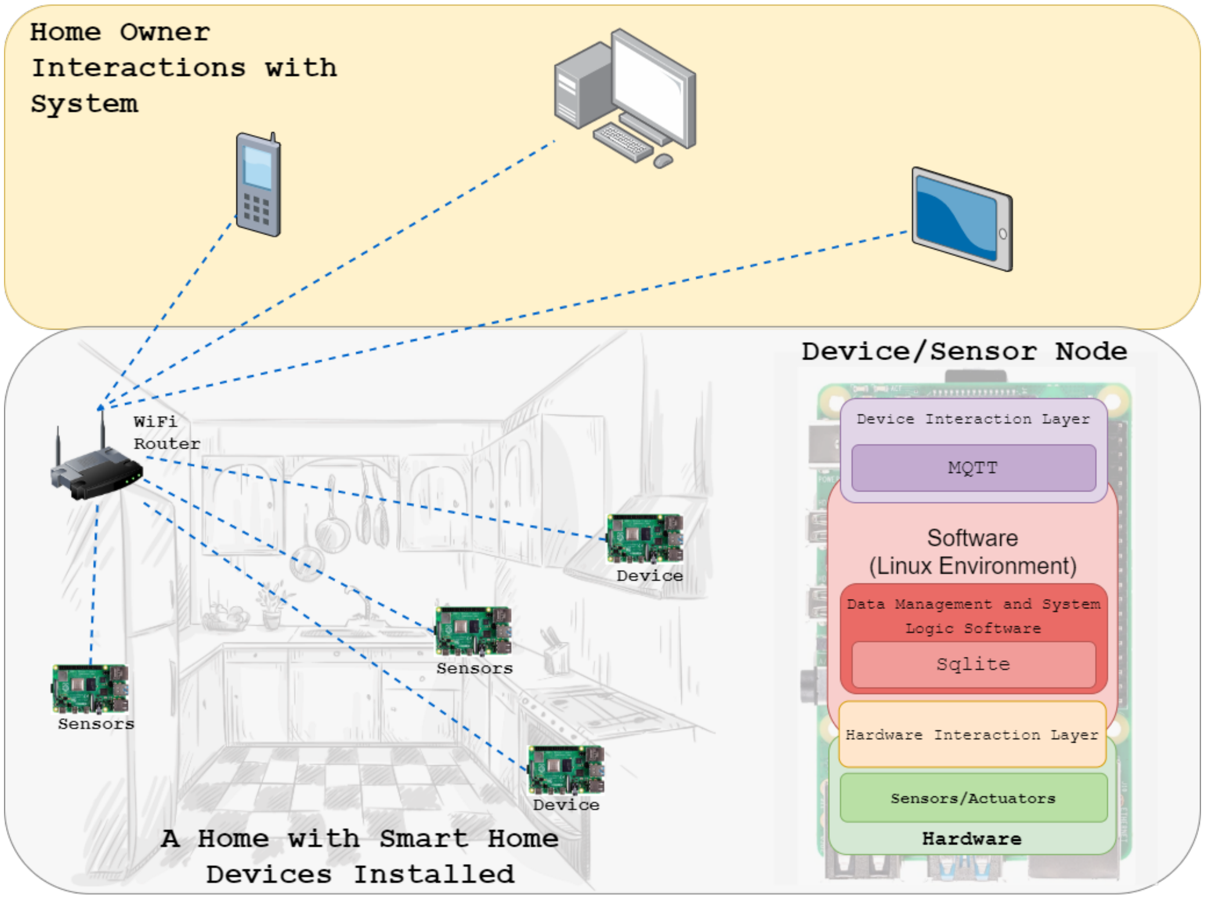
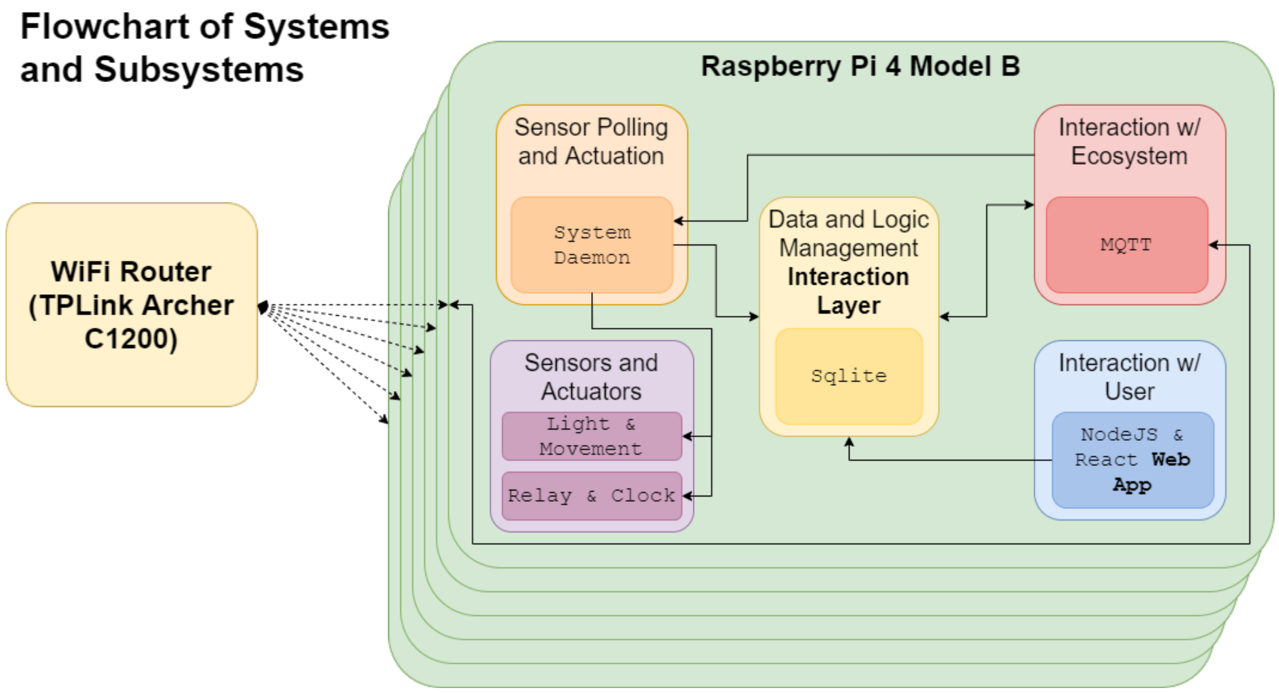

Home Owner Interactions with System
WiFi Router
Device
Sensors
Sensors
Device
A Home with Smart Home Devices Installed
Device/Sensor Node
Device Interaction Layer
MQTT
Software (Linux Environment)
Data Management and System Logic Software
Sqlite
Hardware Interaction Layer
Sensors/Actuators
Hardware

Figure 1: Overall System Diagram



IV. Design Trade Studies

A. *Hardware Platform of System*

Limitations:

1. Hardware platform needs to be able to host MQTT broker as per interaction layer
2. 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.

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:
   1. IP address
   2. Last seen heartbeat
   3. Serial number
   4. Whether it is a master node (hosting webapp or broker)
2. Network ID
3. Defined interactions
4. Sensor data

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 few 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 exists on all 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 1-3 would be defined in a shared database, and item 4 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 1-3.

Our reasoning behind this is as follows: items 1-3 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 1-3, and have sensor nodes also store item 4. Whenever 1-3 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 the 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.

V. System Description

A. Hardware Systems

The hardware system is made up of a smart alarm clock, smart coffee pot, a smart bulb, and three sensor devices. All devices will be built on top of the Raspberry Pi platform. The devices will communicate over wifi using a consumer wifi router.

1) Smart Alarm Clock/Coffee Pot/Bulb

The smart devices will be built using a solderable raspberry pi hat that will connect to the gpio pins on the raspberry pi. These will be simplified smart devices to simulate realistic smart devices used by consumers today. The smart coffee pot, in particular, will be created by "hacking" a consumer coffee pot with a relay switch and controlling that relay with a raspberry pi.

2) Sensor Devices

The sensor devices will be created using a solderable hat on the raspberry pi. The device will have light and motion sensors, and if it is easier to get those packaged with other sensors, other sensors as well.

B. Interaction Layer Systems

1) Device interactions

Interactions will be managed using MQTT. The master node in the system will run the broker, and all sensors and devices will connect as clients. Sensors will publish to sensor-specific topics, and devices will subscribe to topics required by user-defined interactions.

Note that the broker process may be running on any device or sensor, *including* potentially the device or sensor involved in the transaction. Sensors will 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 will publish this information to an mqtt topic of the form "[device_serial_number]/[type_of_data]". Devices with interactions tied to a sensor will 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,
   'Type_of_data' : string,
   'payload' : json (specific to the type of data being
transmitted)
}
```

2) Local device storage

Devices will store config data, heartbeats, and sensor data in a local SQLite database.

3) Hosting the Webapp

See "master selection" below. A master node will be elected, and that node will host the webapp. Users can access the webapp to interact with the system.

4) 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.

5) 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 will be implemented on top of our MQTT layer. All nodes will subscribe to a topic named "config_updates". Whenever there is a config change, the master node will publish to this channel, and the devices will update their version.

6) Device Heartbeats

This will be implemented on top of our MQTT layer. All nodes will publish to a topic of the form "[device_serial_number]/heartbeat". The master node will subscribe to these topics, and receive heartbeats from the nodes. It will then publish the collection of heartbeats, along with its own heartbeat, to a "heartbeats" topic. All nodes are subscribed to this topic, and will update the information locally. While this introduces a level of redundancy, it makes it so that if a master fails, all nodes are aware of other nodes in the system that are up, so that they can communicate to elect a new master.

C. Webapp Systems

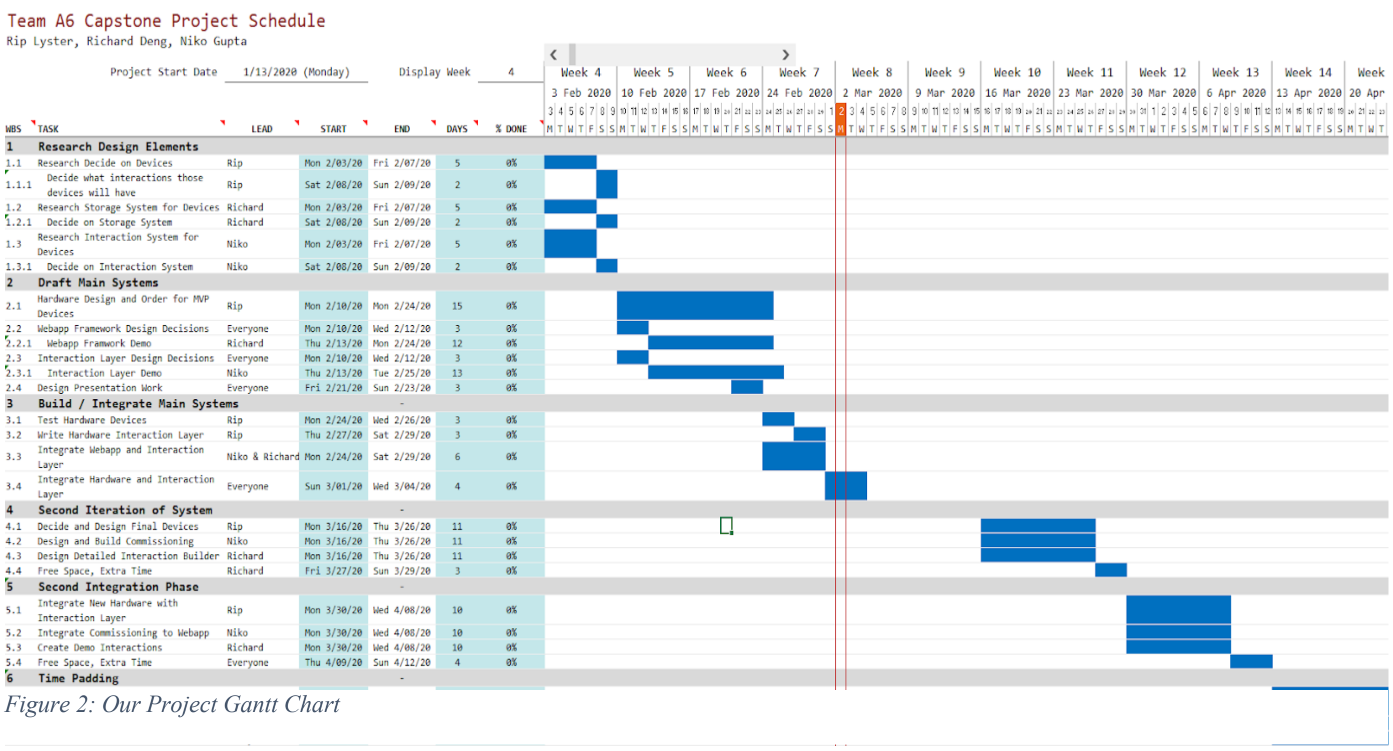
The frontend will be built using React. Light, fast, and easy to prototype with, React can also be containerized easily using Docker, making it extremely portable.

1. Devices
2. Interaction
3. Registering devices
4. Registering interactions
5. Sign up

The backend of the web application will be built using ExpressJS. Simple, lightweight, and minimal, ExpressJS 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.

Communicates to system via REST API. Mounted on Raspberry Pi using a Docker container.

VI. Project Management



Team A6 Capstone Project Schedule

Rip Lyster, Richard Deng, Niko Gupta

Project Start Date 1/13/2020 (Monday) Display Week 4

Week 4 3 Feb 2020 | Week 5 10 Feb 2020 | Week 6 17 Feb 2020 | Week 7 24 Feb 2020 | Week 8 2 Mar 2020 | Week 9 9 Mar 2020 | Week 10 16 Mar 2020 | Week 11 23 Mar 2020 | Week 12 30 Mar 2020 | Week 13 6 Apr 2020 | Week 14 13 Apr 2020 | Week 20 Apr

WBS	TASK	LEAD	START	END	DAYS	% DONE
1	**Research Design Elements**					
1.1	Research Decide on Devices	Rip	Mon 2/03/20	Fri 2/07/20	5	0%
1.1.1	Decide what interactions those devices will have	Rip	Sat 2/08/20	Sun 2/09/20	2	0%
1.2	Research Storage System for Devices	Richard	Mon 2/03/20	Fri 2/07/20	5	0%
1.2.1	Decide on Storage System	Richard	Sat 2/08/20	Sun 2/09/20	2	0%
1.3	Research Interaction System for Devices	Niko	Mon 2/03/20	Fri 2/07/20	5	0%
1.3.1	Decide on Interaction System	Niko	Sat 2/08/20	Sun 2/09/20	2	0%
2	**Draft Main Systems**					
2.1	Hardware Design and Order for MVP Devices	Rip	Mon 2/10/20	Mon 2/24/20	15	0%
2.2	Webapp Framework Design Decisions	Everyone	Mon 2/10/20	Wed 2/12/20	3	0%
2.2.1	Webapp Framwork Demo	Richard	Thu 2/13/20	Mon 2/24/20	12	0%
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%
4	**Second Iteration of System**			-		
4.1	Decide and Design Final Devices	Rip	Mon 3/16/20	Thu 3/26/20	11	0%
4.2	Design and Build Commissioning	Niko	Mon 3/16/20	Thu 3/26/20	11	0%
4.3	Design Detailed Interaction Builder	Richard	Mon 3/16/20	Thu 3/26/20	11	0%
4.4	Free Space, Extra Time	Richard	Fri 3/27/20	Sun 3/29/20	3	0%
5	**Second Integration Phase**			-		
5.1	Integrate New Hardware with Interaction Layer	Rip	Mon 3/30/20	Wed 4/08/20	10	0%
5.2	Integrate Commissioning to Webapp	Niko	Mon 3/30/20	Wed 4/08/20	10	0%
5.3	Create Demo Interactions	Richard	Mon 3/30/20	Wed 4/08/20	10	0%
5.4	Free Space, Extra Time	Everyone	Thu 4/09/20	Sun 4/12/20	4	0%
6	**Time Padding**			-		

Figure 2: Our Project Gantt Chart

A. *Team Member Responsibilities*

Richard is responsible for the system's web application.
Niko is responsible for the high-level device interactions.
Rip is responsible for lower level interactions.

B. *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+ One Wire Digital Temperature Sensor	3	5	15
Mini USB Microphone	2	5	10

C. *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.