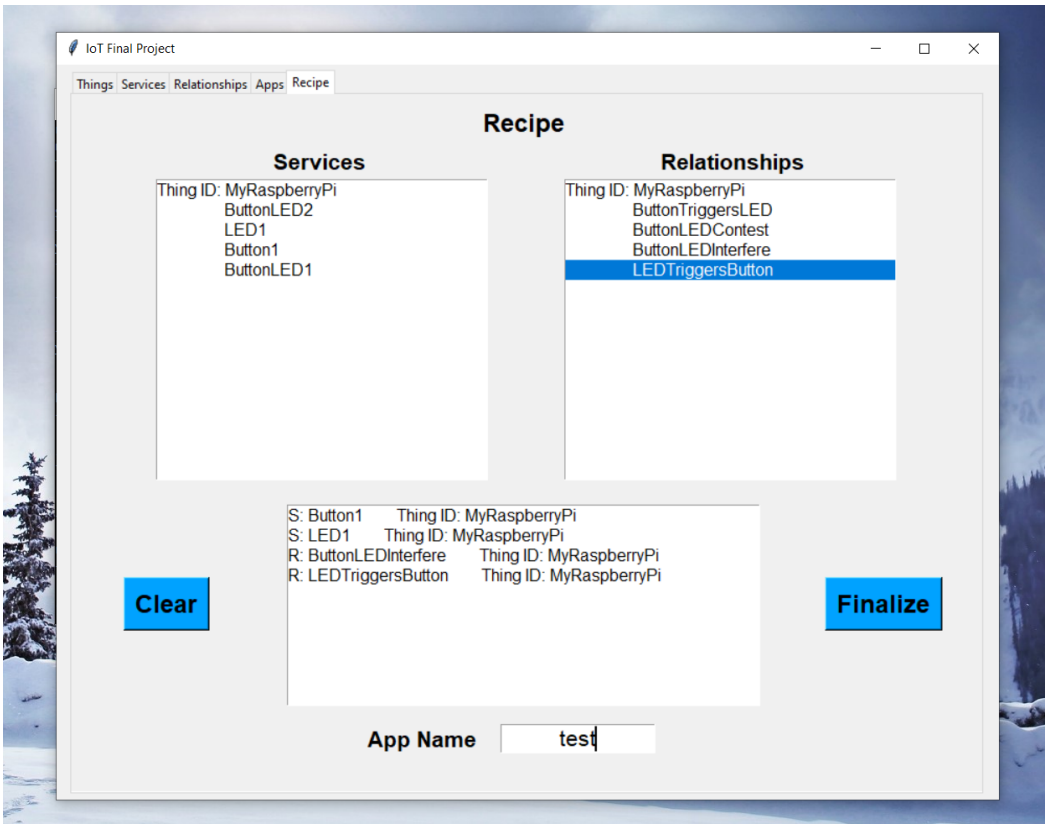
Group # 3

An IDE for Atlas Thing Architecture

Josh Abraham
Alexander Adams
Praveen Anbu
James Horn



1. Introduction

Our application was designed to utilize the Atlas middleware to invoke services and relationships upon different things across different smart spaces under the same network. With the various Raspberry Pi “things” broadcasting tweets on Atlas middleware, our application will be able to scan these tweets to learn about the available services on the network and provide the user with the functionality to view information and create apps based on these services. The user will be able to view the different Raspberry Pi “things” on the network broadcasting Atlas tweets, view the services that these corresponding things provide, and view the relationships that these corresponding things can offer. The user will be able to use the Recipe tab of the application to select and drop services and relationships in order of execution. Then, the user will be able to name and save this ‘app’ that he/she has created. When interacting with the App tab of the application, the user can view available apps, load an app, delete an app, deploy a selected app, and stop app execution. Our interface also provides the user with editing functionality, through which he/she can edit which services and relationships are invoked in each app and edit relationships that may be unbounded. Thus, our application will provide a seamless interface through which a user can interact with Raspberry Pi “things” on the network.

Our project was implemented using Atlas middleware with communication through sockets. Through Atlas middleware, we were able to broadcast tweets about things, services, and relationships through a specific multicast group address (232.1.1.1) and port (1235). Through a Windows computer, we were able to use a UDP socket to conduct multicast scanning in which all Pi’s broadcasting Atlas tweets to the same multicast group and port would be read.

Our GUI was created through Tkinter. We felt this was the most appropriate option after analyzing the Tkinter Python library and realizing its effectiveness in GUI programming. We considered utilizing JavaScript for a frontend interface language but felt that with so many important interactions between the user inputs and communication protocols, it was best for those functions to be intertwined in the Python script. After reading all of the things, services, and relationships through this, the application will update the GUI with the corresponding information. The Things tab will have a list of Things with their corresponding Smart Space. The Services tab will have a list of services organized by each Thing. The Relationships tab will have a list of bounded and unbounded relationships organized by each Thing. The Apps tab will be populated with existing apps already saved from previous sessions of the application. The Recipe tab will be

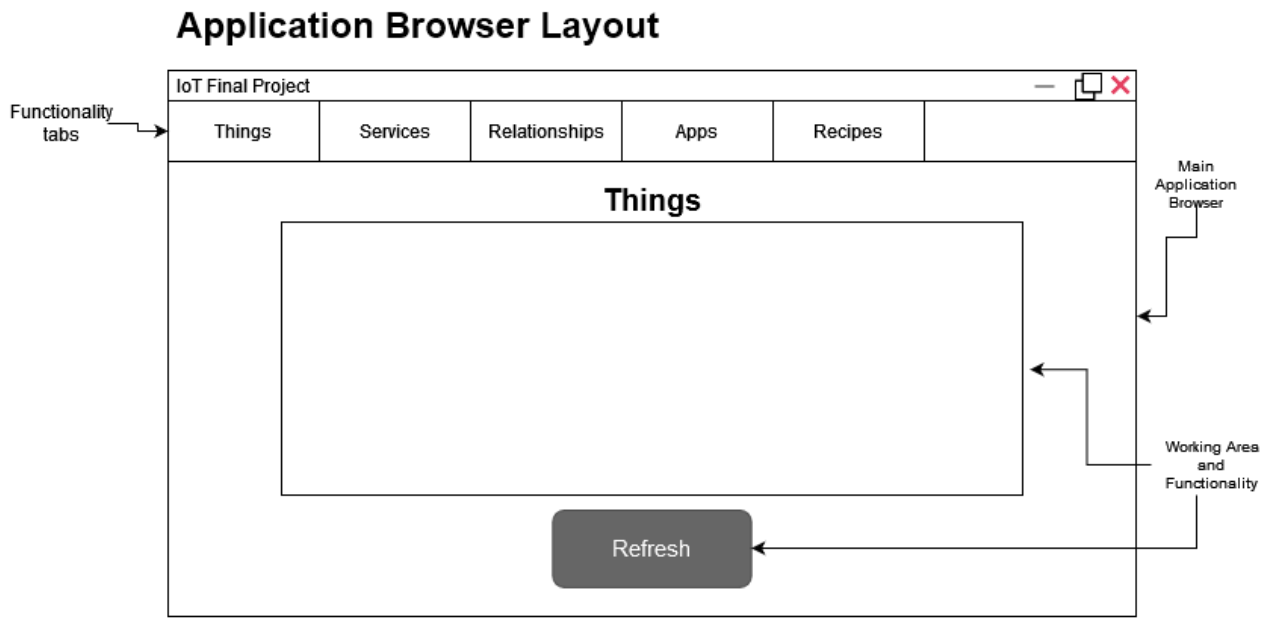
populated with the available services and relationships for combination into creating an application.

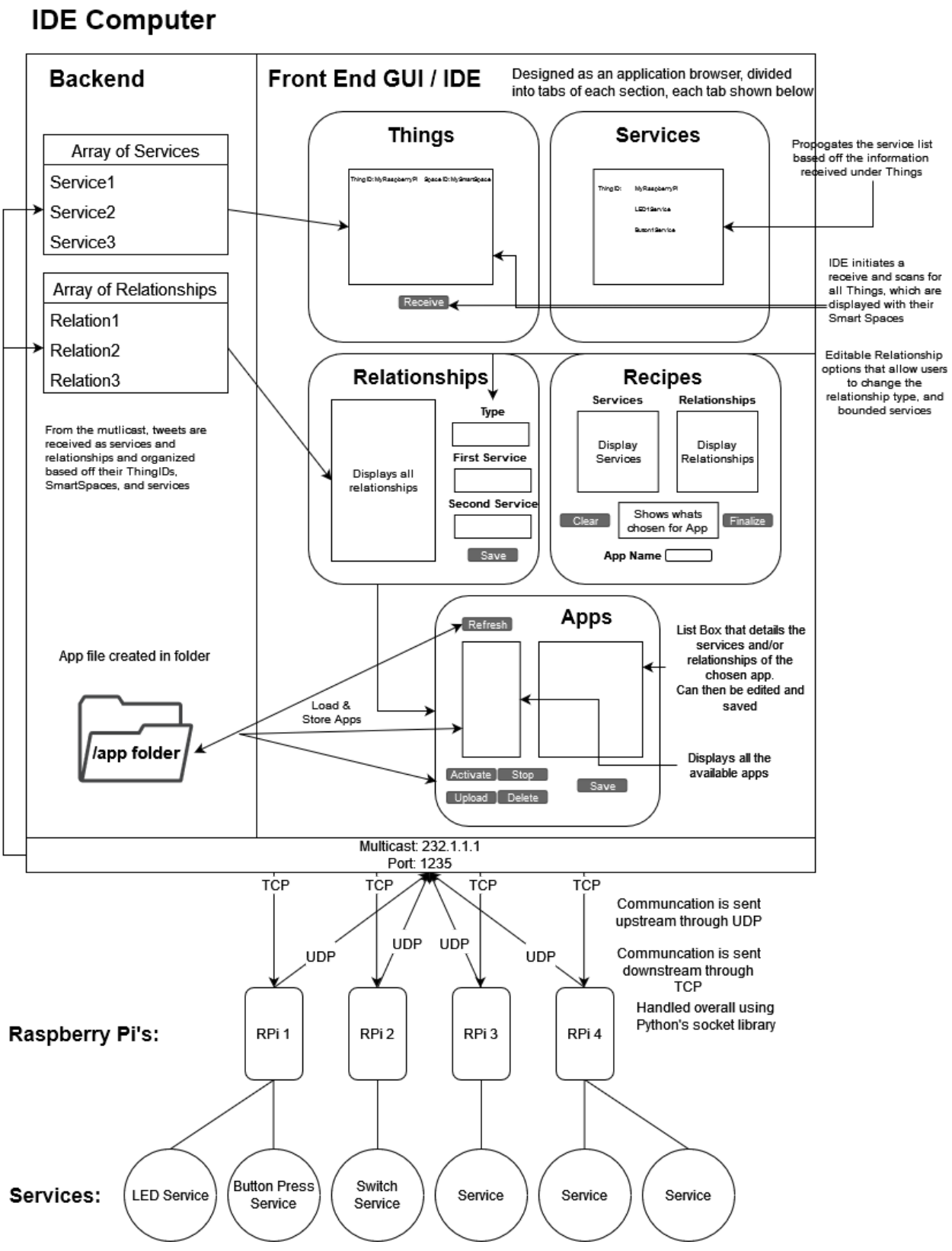
In order to modify and execute these applications, the Apps tab will be used. The user will be able to create these apps through the Recipe tab. By activating an app, a TCP socket will be used to communicate directly with the PI hosting the service. Then, a JSON call with the information found from the service tweet will be used to execute the service or relationship. In order to stop the App upon user command, we have utilized threading to update a configuration file that can allow us to know whether the stop button has been clicked. Apps can also be selected through the Upload button for editing and selected with the Delete button to be removed entirely.

YouTube Video Link: https://youtu.be/feUH7so9ujs

2. Project Design

Raspberry Pi "things" providing services or relationships will be running Atlas and broadcasting constantly. When the user wants to discover things, services, and relationships, they can click the Refresh button on the Things tab of the GUI. This will read tweets being broadcasted for 1 minute and extract the services and relationships. After this it will populate the tabs with each unique option for things, services, and relationships. This can be seen in the diagrams below.




IDE Computer
Backend
Front End GUI / IDE
Designed as an application browser, divided into tabs of each section, each tab shown below
Array of Services
Service1
Service2
Service3
Array of Relationships
Relation1
Relation2
Relation3
From the mutlicast, tweets are received as services and relationships and organized based off their ThingIDs, SmartSpaces, and services
Things
Services
Propogates the service list based off the information received under Things
Receive
IDE initiates a receive and scans for all Things, which are displayed with their Smart Spaces
Relationships
Displays all relationships
Type
First Service
Second Service
Save
Recipes
Services
Relationships
Display Services
Display Relationships
Clear
Shows whats chosen for App
Finalize
App Name
Editable Relationship options that allow users to change the relationship type, and bounded services
Apps
Refresh
Activate
Stop
Upload
Delete
Save
List Box that details the services and/or relationships of the chosen app. Can then be edited and saved
Displays all the available apps
App file created in folder
/app folder
Load & Store Apps
Multicast: 232.1.1.1
Port: 1235
TCP
UDP
Communcation is sent upstream through UDP
Communcation is sent downstream through TCP
Handled overall using Python's socket library
Raspberry Pi's:
RPi 1
RPi 2
RPi 3
RPi 4
Services:
LED Service
Button Press Service
Switch Service
Service
Service
Service

System Architecture Diagram:
https://drive.google.com/file/d/1Mxk7-kCu2bhSkudY1fM9__g0u_Af9g7O/view?usp=sharing

2.1 Challenges Faced

A large majority of our issues revolved around the connection between the Raspberry Pi IoT things and the IDE. An immediate roadblock was a constraint with the multicast reading and the operating system used. Because of certain limitations with Windows, although the ThingIDs broadcasted services and relationships to the default set multicast address and port, our scan function to determine which tweets were being broadcasted was not receiving any messages. So instead, the function was being run on a Linux environment through one of the Pi's temporarily, before we were able to get it working on one of the group member's computer.

Based on our communication protocol using a UDP/Datagram socket, a limitation that arose was the reliability of the multicast and whether any devices sent or received the tweets. Often a case would arise in which a certain important tweet would be tweeting during the time that our multicast read function is reading, but it would not pick up the tweet. Because the multicast could be unreliable, we had to set the scan function to run continuously for at least 60 seconds, to provide redundancy for the tweets which are broadcasted each 20 seconds. Moreover, while testing the Atlas middleware would be rerun multiple times in order to get service tweets broadcasted. So a more robust communication protocol would have expedited the testing and debugging process.

Another of the setbacks was handling the tweets' JSON data type and how it was sent between the Pis running Atlas and the IDE receiving it. Because we were communicating between devices using sockets, and both UDP and TCP protocol, data had to be handled between bytes, JSON, and String objects. After several tests and type conversion implementation attempts, the data was sent as a singular appended String type of all the 'Things' and their respective 'Services' and 'Relationships'. Once received, the string is looped over and split based on a final parameter and delimiter, ensuring that we have JSON-style strings that contain all the necessary key-value pairs. Thus the string is separated into individual strings of the services. These were then converted into JSON types using json.loads(), to pull the important key-value pairs, and inserted into arrays of services and relationships to later be accessed by the IDE. However, this meant duplicate 'ThingID's were sent because services could be a part of the same 'Thing' and 'SmartSpace'. So, each array was checked if it already had a 'Thing' or 'SmartSpace' before an element was added.

3. Implementation

GitHub Project Link: https://github.com/jhorn00/IoTFinal

Front-end:

Python, Tkinter

We made our application using Python and Tkinter for their ease of use. Networking with Python is very simple to implement, and since we needed to communicate with the Atlas devices that made Python a clear choice of language. Tkinter was chosen to create the GUI of the application because we have a group member that has some experience with that library. Additionally, we had a few GUI elements completed in Tkinter previously.

Back-end:

Python, os, socket, json

Python was selected for back-end implementation because it is easy to work with. Specifically, networking is a breeze in Python, so we can easily communicate with our other devices. Data manipulation and file I/O are also very simple in Python. Essentially, many components of our project could be simplified a bit using only standard Python libraries. With the socket library, we can send data between devices with only a few lines of code. With the os library, we can manipulate the application directory. With the json library, we can easily convert socket data to usable objects.

Socket was used for communication between the PIs and the computer hosting the application. As previously done in lab 4, we used a TCP socket to execute JSON calls but we also implemented a UDP socket to read the tweets broadcasted. Sockets provided a simple way to communicate between computers on the same network and have many resources online for help with connecting and sending/receiving data.

The OS library was used to execute the file I/O commands needed in this project. For saving, reading, and editing the available apps, we needed to check our project folder. OS provides a great way to create directories, create files, and edit/write to these files.

The JSON library was used to help store data and execute the JSON API calls. As data would come in as a string, we would convert to JSON as it provides an organized struct to access each

component of data. Throughout the program when working with this data, we would keep it in the JSON format until it is needed for executing an API call.

New concepts:

Multicast, Relationships, User Functionality

Working with multicasting was a new component in this project, as in the previous lab we were specifically working with a socket for direct streaming. However, with many resources online describing the idea and process behind multicasting through UDP/Datagrams we were able to learn and execute it in a fair amount of time.

Working with user functionality in some aspects were new in this project. In order to select services that could be added to a queue for the recipe and edit app files through the GUI were more complex functions of the GUI that required reading different online resources to figure out how to implement.

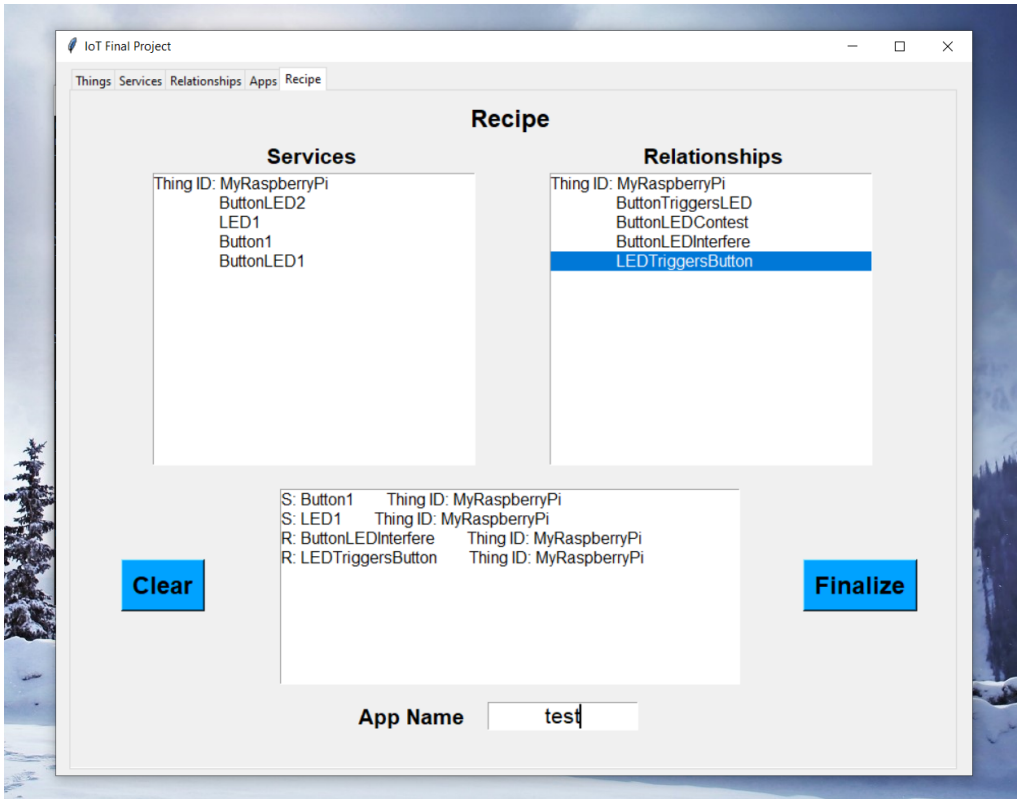


Figure 1. A screenshot showing the Recipe tab in Action

The figure above shows the operation of our Recipe tab. In this tab, we are able to see the Services and Relationships under their respective ThingID. Once we click a service or relationship, it gets added to the current app in order, which we can name in the text box at the bottom. We can clear the queue with the Clear button and save the app with the Finalize button.

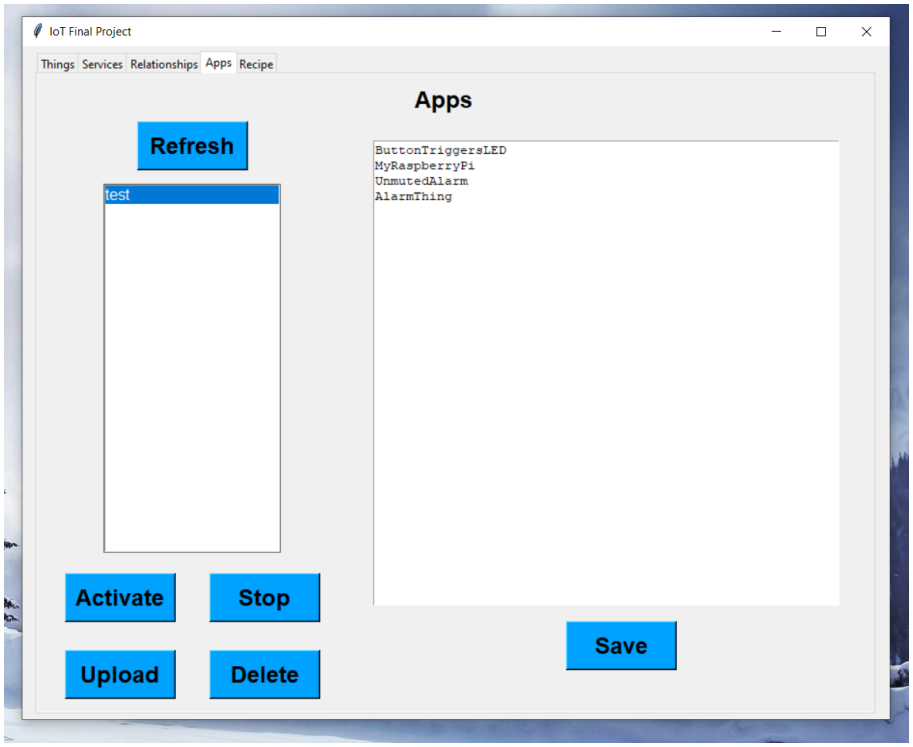


Figure 2. A screenshot showing the App development tab in Action

This figure shows the functionality of the Apps tab. After clicking Refresh, we are able to see the available Apps already loaded through our file system from previous runs. By clicking Upload, we can view the App and edit the app if needed. Then, we can utilize the Activate and Stop buttons to run the app.

3.1 Subjective Evaluation

We thought the project was interesting yet challenging, and we are very proud of our application. We were able to implement most things successfully, and we are most excited about being able to run an app using a recipe. We were able to run both services and relationships in the same app and on different Pi's. The project was somewhat tough to implement, especially since multicasting would work on one Windows system but not others. The scope was maybe a little too much for

undergrads like ourselves, but we were up for the challenge. Not having to implement as many tasks would have been more positive as some things were too much, like having an app manager. Additionally, even with the extension, it was a rush to finish the project as we had to meet up for three solid days to test our code and implement changes.

3.2 Future Work

- We feel that the addition of threading to the service scan would provide a better user experience. Considering our scan happens for a set amount of time (plus or minus a few seconds), a nice progress bar could be created.
- Although we currently have the implementation to edit bounded and unbounded relationships through our GUI, in the future we could also implement a feature that allows users to create relationships from scratch through the GUI. This would be relatively simple to implement as it builds off the same concepts as creating an App.
- We would have liked to have the information from a loaded app transfer to our Recipe page rather than the App page. With the Tkinter library it is difficult for us to migrate information between our tabs. It is technically possible, but would take a great deal of effort for relatively little gain. In its current state, our system still allows loaded apps to be edited, it just requires typing the changes.

3.3 Distribution of Effort

Student Name	What was done?	% Effort	Comments
Josh Abraham	Worked on some of the services and Atlas implementation, input for gui code, system architecture diagram, documentation, and video editing	25%	The project would have been more understandable if the requirements and features needed were more detailed and how to implement them, but it provided an interesting opportunity to attempt
Alexander Adams	Handled the Frontend. Made the Tkinter GUI. Made sure all the buttons worked and listboxes were correctly filled out	25%	Interesting project but needed a little more time to provide an amazing product

Praveen Anbu	Worked on relationship handling code,code for invoking services, testing of code through Atlas, and knowledge transfer of lab specifications.	25%	As an undergrad student, I felt this project was quite challenging when experimenting with different communication protocols. However, once that was figured out, it was easier to implement functionality.
James Horn	Handled application files, worked on relationship handling, did some of the C++ services, documentation	25%	I wish we had a bit more time, but we managed with what time we got. In the future it might be better to provide some form of example to demonstrate the more complex requirements of the project.