Senior Project Design Report

Spirit: A Home Automation System

Andrew Choi

Advisor:
Professor Lynne Slivovsky

06/11/14

Table of Contents

Introduction (Project Charter)

Project Overview

Spirit is a multi device home automation system. Designed and implemented for California Polytechnic State University's Senior Project program, this project was worked on during the duration of two school quarters from January 7, 2014 to June 13, 2014. The system consists of monitor/controllers designed to carry out everyday tasks in the average American household and an accompanying mobile application designed to receive information and control the devices. The monitor/controllers, or "Spirits", are primarily developed using Arduino development tools and Arduino microcontroller boards. The spirits include a thermostat, named *Tempus*, an electrical outlet, *Electrus*, and a wall light switch, *Luxos*.

Client

This project was created designed and developed by Andrew Choi. There are no third party owners, clients, developers or designers. All production was carried out without the intention of profit. All intellectual property of the project is retained by Andrew Choi and future decisions regarding the project are subject to his judgement.

Stakeholders

As there are no clients, stakeholders of this project are minimal. Some stakeholders would be companies interested in acquiring the intellectual properties of the system. Others would include any user desiring a home automation system. As the main purpose of the project is to better a resident, they will be affected by the success of this project. Any company relating to home and office heating or electricity will be affected by this project as well, as they might install, maintain or recommend the project.

Framed Insights and Opportunities

There are many needs for the project. Some of these needs can be laid out into tiers of functionality of the project.

Tier 1: The minimum viable product. This would be the simplest form of a WiFi enabled thermostat. It would consist of an Arduino Uno that could turn a furnace on and off. The Arduino would be connected to a home network by use of a WiFi shield, and the user could turn the furnace on and off using a simple button based interface on a web browser.

Tier 2: Temperature sensing thermostat. This would be added so the user could set a temperature using the web interface, and the Arduino would automatically turn the furnace on and off based on the temperature of the buildings climate.

Tier 3: Mobile application. This step would involve making the user interface an Android native application with a easy to use and appealing UI so the user could be mobile while on the home network. The user should also be able to see see what temperature the inside climate is via the app.

Tier 4: Remote access. This would extend the usability of the system to global access. Connecting the Arduino to a backend server the the Android client can access would allow users to control the thermostat from an internet connection not necessary within the network that the thermostat is connected to. This is one of the most important pieces of the project, as control of the thermostat remotely would appeal to most users for many critical reasons.

Tier 5: Individual user access. In terms of security and scalability, individual user accounts would need to be implemented. This would allow users of the Android application to login using user credentials and access their own personal thermostat, without anyone else having access to it.

Tier 6: Other monitor/controllers. Expanding the concept of the home automated thermostat to other devices would enable a whole system of controllable monitors for the household. These would include a lightswitch and an electrical outlet.

Goals and Objectives

Goals:

- User control of a thermostat
- Wireless connectivity
- Easy to use and appealing UI
- Android application as a UI
- Remote access
- User accounts
- Expansion to other devices

Objectives:

- Connect temperature sensor to the Arduino
- Communicate between WiFi shield and Arduino
- Setup Arduino to be on a home network
- Have Arduino relay the temperature over WiFi
- Program the Arduino to turn the furnace on
- Allow a user to set a temperature for the thermostat wirelessly
- Develop an Android app as the end interface for control of the system

- Create a backend server to extend access to remote networks
- Connect Arduino thermostat to backend server, having the server trigger actions
- Connect the Android client to the backend server, having the app trigger actions
- Allow access to the backend server only through accepted user credentials
- Create similar devices that carry out other tasks that will be controllable as well

Outcomes and Deliverables

The result of this project should have two parts: hardware that is the thermostat and software in the form of an Android app. The hardware portion of the project could also include other devices. The app should be downloadable by users via the Google Play store. Deliverables for the project will be documentation of the entirety of the project, and a report completed as by specifications of the Cal Poly Senior Project requirements.

Related Work

As home automation becomes a more highly desired product, many companies and businesses are coming up with solutions for this void of a field.

Nest, a product and company acquired by Google, is a frontrunner in the area of smart thermostats. They have an application for remote access of the thermostat, as well as an AI that predicts the temperature at which the the environment should be set.

Wemo is a series of product from the company Belkin, whose biggest seller is a WiFi enabled outlet. These devices come with an application that controls them as well as receives energy usage information from the device.

In more recent news, Apple recently unveiled their new idea for home automation. It involves new development software that can access home automation devices, all completely implemented in iOS 8, including Siri control.

Formal Product Definition

The goal of this project is to create a home automation system with remote user access through an Android application as a user interface.

Marketing Requirements

- The thermostat should be remotely controllable
- Users can access the thermostat through an Android application
- Device control should realtime
- The application should be free to use

Engineering Requirements and Justifications

- Hardware development is Arduino based
    - Upon completion, development on a larger scale would be conceivable through this development process.
- The Arduino internet connection is over a home WiFi network
    - Most users will have a WiFi network set up.
- The Arduino will run off of one 9V battery
    - These batteries are easily accessible and replaceable.
- The Arduino will make use of a temperature sensor
    - This will allow for automatic turn on and shut off depending on environment temperatures.
- The Arduino will be able to control a standard 24V furnace input
    - Most home heating uses this technology.
- The Android application conforms to Google Play terms
    - Not following these terms would result in an exclusion from the Google Play Store.
- The Android application runs on devices using Android Honeycomb or later
    - This will ensure a large compatibility with Android devices on the market.
- The Android application runs on tablets as well as handheld devices
    - This will enable users of all types of android devices to operate the application.
- The system will use a backend server
    - Use of a backend server enables remote network access.
- User credentials will be used to login to a specific account
    - This prevents unauthorized access as well a personalization.
- Total cost of hardware will be under $100
    - Development cost transfers to the customer, so costs should be kept to a minimum.

Criteria

The final design of the *Spirit* system will be determined based on the following criteria:

Criteria	Units	What's better
Speed	seconds	faster
Availability	seconds	less (downtime)
Power	Watts	less
Ease of use	steps	less steps
Visually appealing	ratings	higher ratings
Cost	$	less $

The User Experience

This system is meant to be simple and easy to use. Use of the system will consist of two parts: setting up a device, and using a device. Both parts should be easily doable without instructions.

Setting up a device starts with the Android application. After logging in with a user account, the user is taken to the main Spirit screen which is a display of the devices currently controlled by the account. To add a new device, the user presses the add button in the upper right corner. This starts the Add New Device process. At first, the user sees the New Device information page where they can fill out fields about the device such as type (which *Spirit* is being added) and name (such as “bedroom light”). Here the user also enters the WiFi network name and password (the network name should auto fill). Once the user presses “next”, they are prompted with a dialog telling them to press and hold the configure button on the device. If bluetooth is disabled, the user is prompted with a dialog allowing them to enable it. The application then automatically connects to the device, delivering all necessary information for the device to connect to the internet and be controlled by the account of which the app is currently logged in. The app then returns the user to the main Spirit screen where the new device is now seen.

Using a device is relatively straight forward. After logging in, the user is presented with the apps main screen. Here, the user selects one of the devices controlled by their account. Once pressed, the app shows a screen, specific to the device, from which a user can control the device.

In both parts of the user experience, if a user is using a tablet Android device in landscape orientation, the app is present as two screen, one left and one right. The left is always the main Spirit screen, where one selects a device to control. The right screen shows the secondary task, whether that be the process of adding a new device or controlling a certain device.

Personas

The majority of users of this system will be any person with a home WiFi network and a common living residence. This person could be a homeowner, an apartment renter, or a family member of whomever they live with. The user will not need any exceptional technical skills or knowledge of electricity, wiring or hardware. They will, however, most likely be well acquainted with mobile applications, and should not have issues using this one.

Another type of user that this system may utilized by is a Building Maintenance Supervisor or Engineering Supervisor of an office building. With no limitation on the number of many devices that can be added to an account, and with plans for expansion of new devices, the Spirit system would work well in an office building scenario. This supervisor would have access to all heating and cooling of different floors, as well as control of lighting and power station as specific as desk to desk.

Use Cases

The main use case of this system is to automate a home or building. More specifically, use cases are controlling a certain device in ones home or building. A user turning on a light can now be done from their phone in another room in another city on the other side of the world. Any need for visual control of a device, or a need for remote access to a device can be done through this system.

Design and Justification

Overview

The whole concept of the Spirit system is based on remote access of a device. However, this does not include just a device and its controller. There are many pieces that all have to work in conjunction in order for an action to happen. This section describes the nuts and bolts of the system.

System Architecture

In order for all requirements and goals of this project to succeed, many components came into play during the design and implementation of this system. Security of accounts was a big issue, one that could not have been dismissed, but also one that added much more design and implementation problems.

First, in trying to comply with a real time expectation of control, many services were considered to be used in this project. PubNub, a realtime network, was chosen. This brought in much more coding and research needs. However, once understood, the process was simple. For security reasons though, The PubNub network could not stand alone. PubNub can connect a controller and a device with authentication, but another server is needed to administer the authentication for user accounts.

For the backend server, after more research and consideration of many hosts, Google App Engine was chosen to handle all account information and authentications. Once this server was set up, Authentication tokens could be set in the realtime network and then given to the controllers and devices to communicate solely over the realtime network.

Once all the networking and backend server were set up, all that was needed where the actual devices and controllers. The devices were implemented using Arduino hardware and developed in C using the Arduino software. The controllers were implemented as an Android application, using standard Android SDKs from google.

Once set up, the system works as follows:

1. Through the Android app, a user signs in with user credentials
2. Ther Google App Engine server generates an authentication token for the realtime network if it doesn't exist
3. The server enables communication based on the user and authentication token
4. The server sends the authentication token to the app
5. The app gives the realtime network channel and authentication token to the device during configuration
6. Using the user credentials and the authentication token from the Google App Engine server, the Android app and device can now communicate through the realtime network

These steps can be seen visually in **Figure 1**.

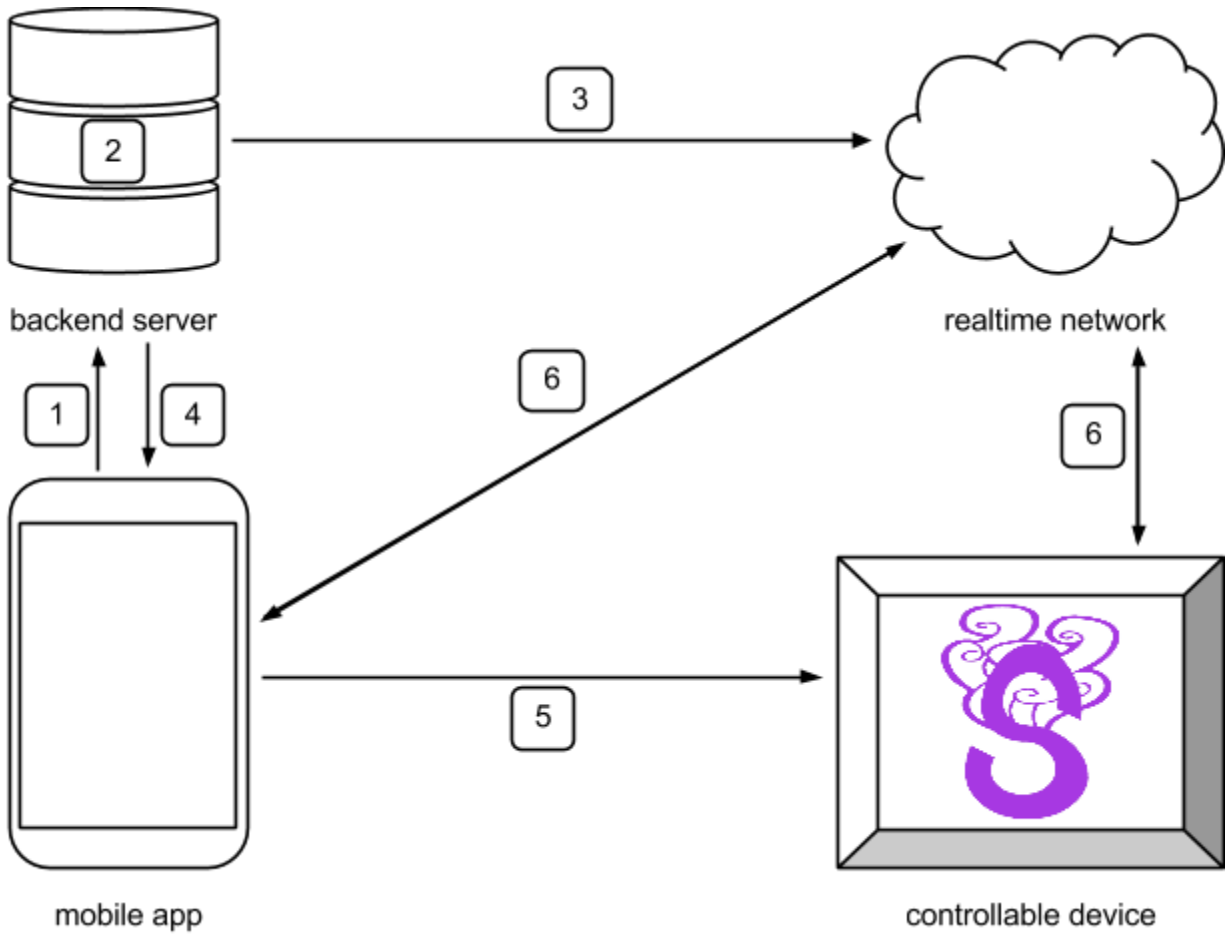

Figure 1: System Architecture

The Google backend server does not create a new authentication token every time a user logs in (forcing the app to access the server). The token is created once for the configuration of a device, leaving the authentication the same so the device can send data without relying on the app configuring its authentication every time.

Hardware Architecture

Hardware for this project started simple, and kept progressing as new features found their way onto the drawing board. In the initial design, an arduino would access a home network using a WiFi module, and that was basically the extent of it. With the addition of authentication and remote access (in turn adding servers), more needed to be added. The need of direct communication from the Android application to the device before WiFi connection could be established caused a bluetooth module to join the mix of hardware.

The development board used for this project was an Arduino Mega 2560 R3. Originally, an Arduino Uno R3 board was used, but due to the lack of space needed for the bluetooth, WiFi and sensor module libraries, the higher flash capacity Mega 2560 was needed.

Hardware connections are as seen in **Figure 2**.

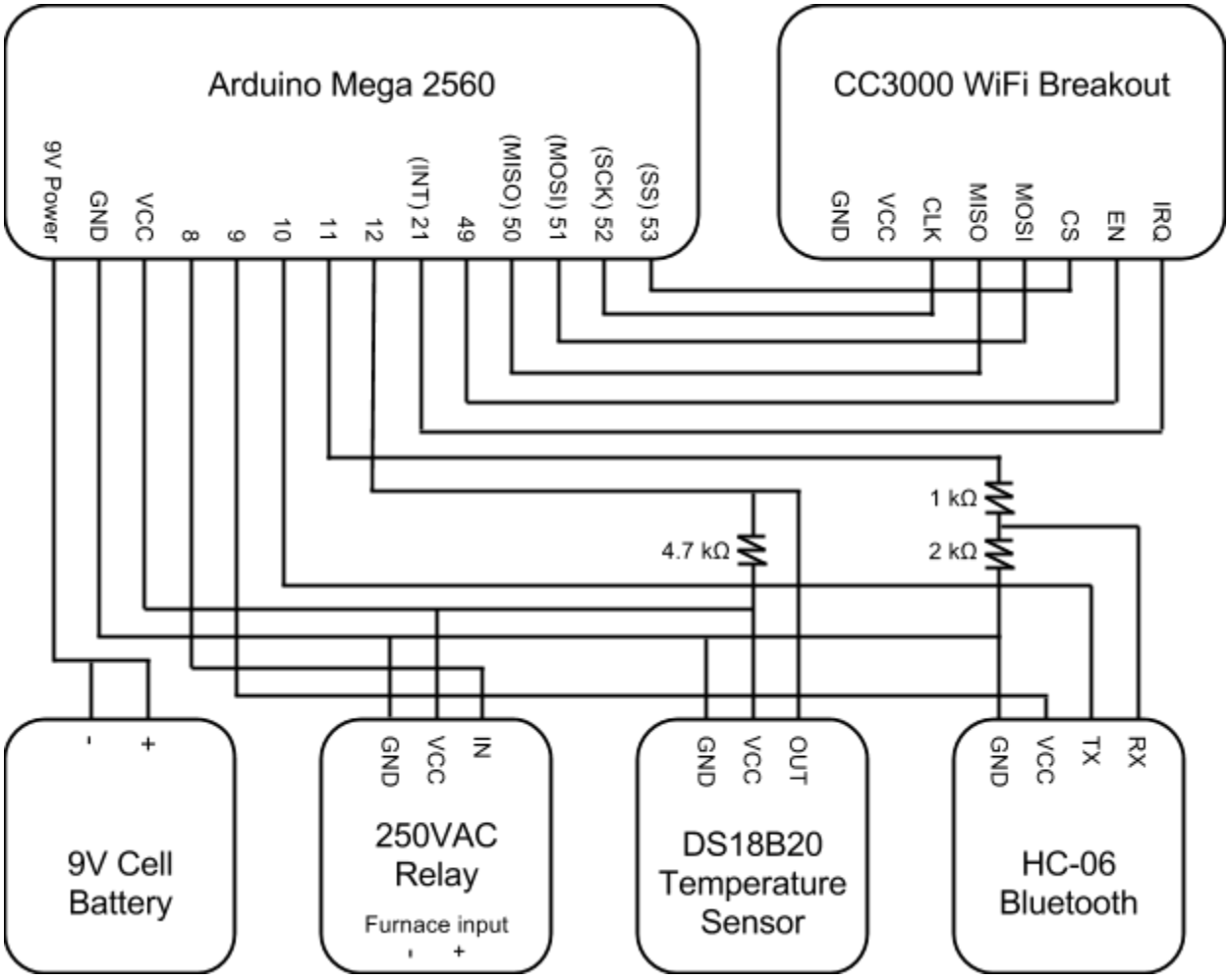

Figure 2: Hardware Architecture

Since VCC of the bluetooth module is connected to pin 9 of the Arduino, through code, the power to the module can be turned on or shut off, depending on if the device is in “config mode” or not. This was extra power used by the module is only drawn when necessary. The input and output of the bluetooth module uses Serial (designated at pins 10 and 11) to communicate with the Arduino. The temperature sensor is digital, using the Arduino included OneWire library. The relay that controls the 24V drop across the furnace control wires has one control input, so basically pin 8 of the Arduino controls the furnace. The Texas Instruments made WiFi module takes advantage of the Arduino’s SPI connection, and interrupt pin and an enable pin.

This is the hardware for the thermostat Spirit device *Tempus*. Hardware for the other devices alters slightly, namely an additional input button acting as the light switch for *Luxos*, and no temperature sensor for *Luxos* or *Electrus*. Other hardware designs had been started during the course of this project. A Spirit device named *Inpedus* is a door lock. It uses a servo motor to alter the lock position. It is designed to fit on top of standard deadbolt locks, and would allow a user to change the state of the lock manually as well. Other devices *Aquos*, and irrigation system controller, and *Proximus*, a proximity alert device, were in early stages of concept development.

Software Architecture

There are three main parts for this project in terms of software. There is the Android application code, the Arduino device code and the Google App Engine server code. All three parts interact in unison for the system to work.

The “Spirit” Android app makes use of the relatively new Android development style, fragments. Fragments are used in this app so that there can be two panes in the landscape orientation of tablet devices. For this, there is one activity, MainActivity, that controls all of the fragments. This activity actually has two xml layout files, activity_single_pane.xml and activity_double_pane.xml, that get chosen by the Android device depending on its screen size, as seen in **Figure 3**.

Figure 3: Master Detail single pane and double pane

The first fragment to open is the SignInFragment (**Figure 4**). Once a user presses the sign in button a Google account login dialog pops up as seen in **Figure 5**. If the Android device already has Google accounts linked with it, those accounts automatically show up in the dialog, allowing the user to choose one without entering account credentials. If no Google accounts are linked with the device, the user is prompted to sign in with a new one, or register for a Google account. Google accounts are used for this system due to security and simplicity from a user standpoint. Future designs might include implementations of facebook login as well as email registration. If the DialogActivity has an authenticated user upon the activity result, the user is considered successfully logged in.

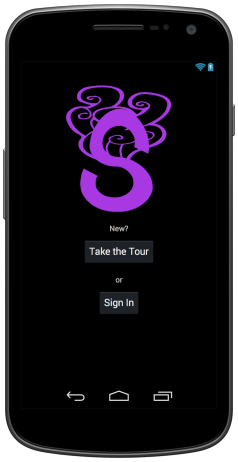

Figure 4: SignInFragment

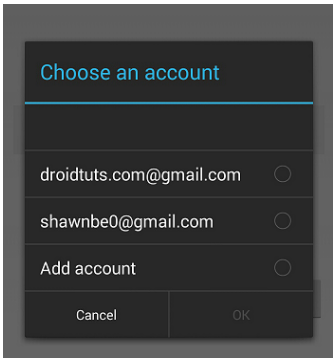

Figure 5: Google Sign In (image from http://www.shawnbe.com/)

Once logged in, the MasterFragment is shown (**Figure 6**). This is the view of all of the users Spirit devices. The fragment has a “sign out” button, a “+” button and a scrollable GridView with a custom adapter to specify views for each device. If in landscape orientation of a tablet device, the MasterFragment appears as the left third of the display, leaving the right to open another fragment.

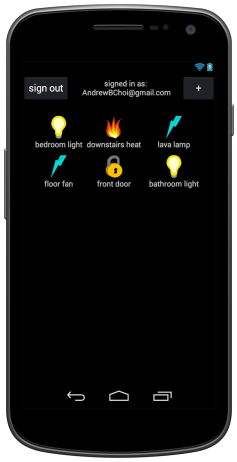

Figure 6: MasterFragment

If the “+” button is pressed, the AddSpiritFragment is shown (**Figure 7**). Again, if in landscape orientation of a tablet, the master flow detail is taken into effect, and the AddSpiritFragment is shown in the right portion of the screen. In this fragment, there is a GridLayout (adapted from the support library for earlier android versions) that contains all the necessary fields to fill out for configuring a new Spirit device. These fields include text fields as well as a spinner to choose which device the user is configuring. Once a option in the spinner is chosen, a updated icon of the device appears next to it. In this fragment, there are also “cancel” and “next” buttons. Once the “next” button is pressed, the app accesses the Google App Engine backend server to receive an authentication token for the channel of the realtime network. Once this is recieved, a custom DialogFragment appears telling the user to press the Spirit device config button. After the dialog is dismissed, a bluetooth connection is created using the BluetoothAdapter class. This sets up a connection based on the bluetooth information of the device already in the app. There should be no more prompts for the user, as the app automatically finds the device, sets up the connection and transfers the data. The app sends WiFi network information, user account information and the realtime network information. Upon a successful transfer of data, the device, along with all information of it (name, type of device, realtime network channel, etc.) are added to the list that the custom adapter of the GridView in the MasterFragment uses. The MasterFragment automatically updates with the new device, and the AddSpiritFragment detaches from the MainActivity, going back in the fragment stack to the MasterFragment.

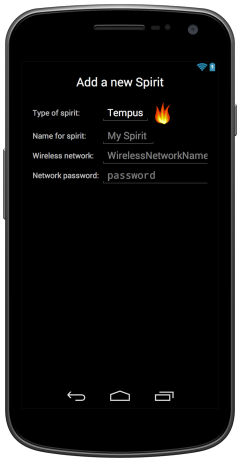

Figure 7: AddSpiritFragment

If an item in the GridView of the MasterFragment is selected, a specific fragment is started depending on the type of device that the item represents as seen in **Figure 8**. This fragment also will open up on the right side of the screen, leaving the left side as the MasterFragment, if the tablet device is in landcare orientation. Each of these fragments has specific fields based on the type of device, but all of them have a few features in common. The TextView title at the top of the fragment displays the user defined name of the device. The main on/off switch of the device is an the device icon. While in the on state, the icon to turn the device off is greyed out. At the start of the fragment, device information is needed. This information is obtained by making calls to the device through the realtime network. A PubNub instance is created using the PubNub android libraries. The PubNub instance subscribes to the device's channel using the authentication retrieved from the Google App Engine backend server. A publish call is made asking the device for the information (on state, temperature, etc.). Fields are updated as the subscribed PubNub instance receive information. Due to PubNub's high speed traffic, this is all done before the fragment even loads. Once a fiend is changed in the fragment, a similar process is used. The PubNub instance publishes this change to the channel, and once the device is updated and publishes the changes, the fields in the fragment update as well. This is an important way to do this process. If the fields were just updated based on the UI actions, there is no guarantee that the Spirit device received those actions. However, in this method, the fields only change due to responses by the device, guaranteeing that the UI only shows the true state of the device.

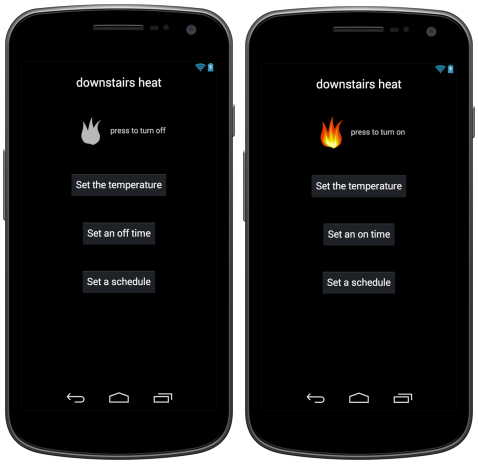

Figure 8: TempusFragment with device in on and off states

While in landscape orientation of a tablet, showing the two pane mode, if a Spirit device is shown in the right, and the user selects another device, the new device's fragment is shown in the right, However, the previous device's fragment is added to the fragment backstack, so if the user presses the Android's native back button, the new device's fragment is removed, and the previous device's fragment is shown. This gives a natural flow to usability. If the device is rotated however, showing the single pane mode, the fragment backstack is cleared except for the current device's fragment. This way the user will return the the MasterFragment (with the grid of all devices) by just pressing back once, not having to travel through all opened devices.

A full software architecture can be seen below, in **Figure 9**.

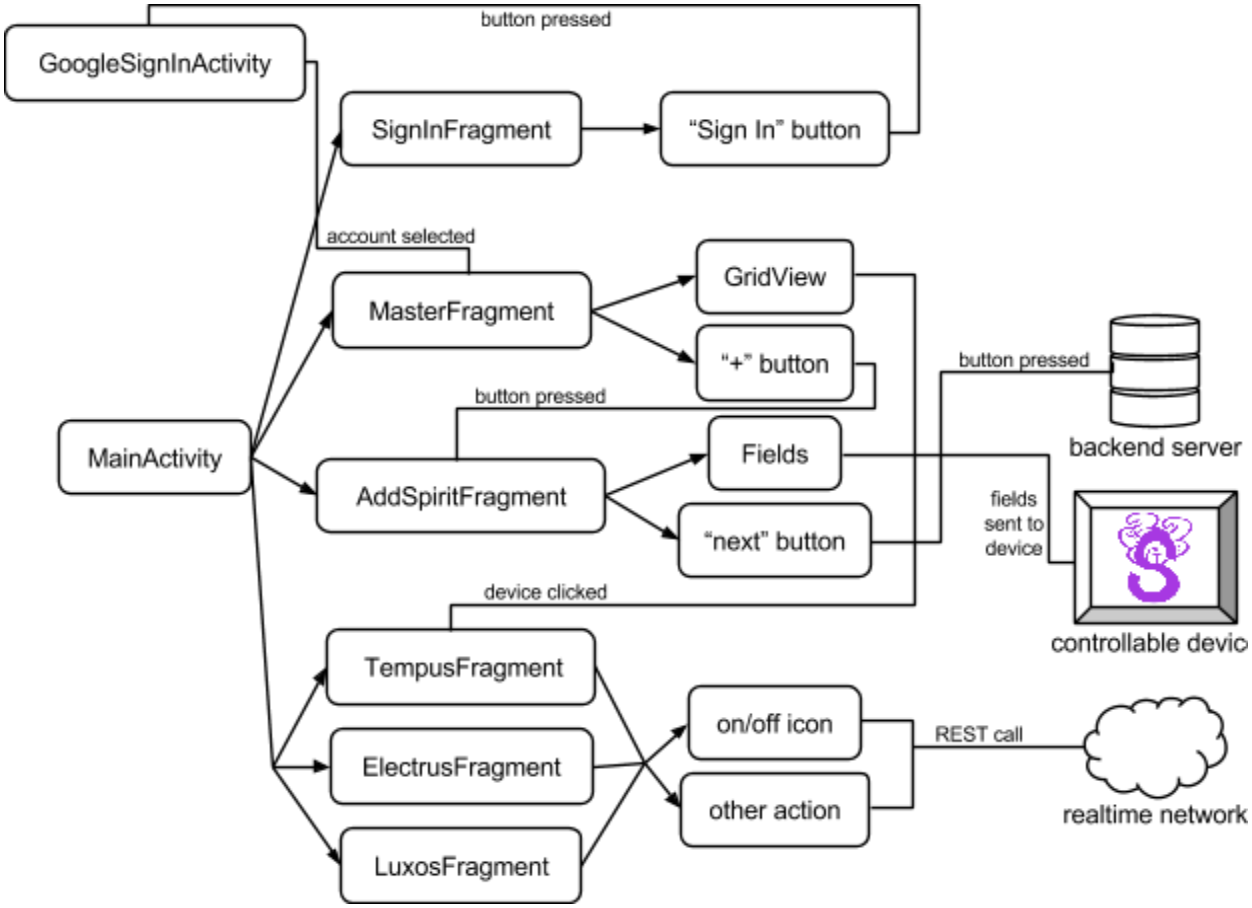

Figure 9: Android Software Architecture

The backend server, implemented using Google App Engine implemented in Java, is used for storing accounts and authentication. It is constantly running on Google servers, therefore allowing 24/7 access from the Android application. The server uses Google Cloud Endpoints for communication with the app. These custom endpoints provide a clean solution to APIs generated by the server.

When the application is setting up a new device and needs an authentication token for the realtime network, it makes calls using the endpoint APIs to the server, passing in a user account and a requested realtime network channel id. The server generates an authentication token using a hashing algorithm to ensure security, and then returns that token to the application. Then, the server, using an instance of PubNub from the PubNub Java libraries, grants the authentication to the channel using a PubNub Access Management call. If the application asks for the realitme network channel authentication token without an authenticated Google account, the server throws an UnauthorizedException. If the application is trying to receive an authentication token for an account and device that has already been set up, the server returns the authentication token already set up for that realtime network channel. This process is shown in steps 1 through 4 of **Figure 1**.

Development for the hardware portion of this system was done using standard Arduino libraries and AdaFruit's CC3000WiFi library. There exist a PubNub Arduino library, but it was forgone in this project due to bugginess as well as lack of functionality needed for optimal control and speed.

Upon startup, the Arduino checks its EEPROM memory to see if it has already set up. This prevents the user from having to set up every time there is a power loss or reset. If the device is not set up, the device waits to be put into configuration mode. Once in configuration mode, the Arduino sets the output pin 9 high, giving power to the HC-06 Bluetooth module. The module blinks a red led until a bluetooth connection with the Android device is made. Once connected, the Arduino waits for serial input from the bluetooth device. The code parses through the input, grabbing a WiFi network name and password, a realtime network channel id and an authentication token. Once all information is correctly retrieved, the Arduino send a success signal to the Android device through the bluetooth module using the serial port. The Arduino then sets pin 9 low, cutting off power to the bluetooth module, saving energy.

After all network information is obtained, or if the device already has contains it on startup, the CC300 Wifi module is setup using the CC3000 library. In then makes a TCP connection to the PubNub ip address. Once a connection is made, the device makes a GET subscribe REST call to the PubNub server, passing in the channel id and authentication token. This TCP client connection then waits until data is received. Once data is received, the information is parsed, looking for a information from the Android app.

If the device receives a request for information from the app, the device makes a new TCP client connection, sending a GET publish REST call to the PubNub server with channel and authentication information.

If the device receives a device state change from the Android app, the device makes the appropriate hardware change (i.e. turn off relay controlling the furnace) and then makes another GET publish REST call with the new device state information through a new TCP client connection.

If there is some change in state of the device, such as the temperature drops below the set temperature of the thermostat or a user presses the light switch, the same process is used to send the device state to the app through the realtime network. Of course, there are plenty of differences between the code for each device. Based on the functionality of each devices, additional tasks are needed, such as recording an turn off time, and periodically checking against that value.

Conclusion

Even though the initial idea of this system was to have a simple thermostat that could be controlled on a home network, all of the additions throughout the course of this project were enjoyable to research and implement.

The change of the UI from a simple web view to a native Android application was desirable to me since I have taken Cal Poly's Mobile Application Development course and used that knowledge in my Capstone project. However, throughout the project, I learn much more about Android development than in any of these other courses. Google's Android templates provide a Master Detail flow, but it is only ever for a list view. Wanting a easy on the eyes grid of devices, I decided to look into changing the fragment based activity to incorporate this. Researching online showed no examples of this being implemented before, so I was thrilled when I figured it out myself. Many features that were new to me came into play also. Bluetooth was a struggle, and using broadcast receivers and connection threads were foreign to me at the start of the project, but now I am comfortable using them.

The decision to allow access on remote networks cause the most amount of addition work and frustration, but also provided an amazing learning experience. Starting off by implementing an Amazon AWS EC2 server and then switching over to a Google App Engine server gave me experience with multiple types of backend servers that can be used with applications.

Originally, the realtime network I was using for quick communication between the Android app and the Arduino was LogMeIn's Xively network. It worked well, but after some development and further research, I found that PubNub was more suited to my project, as they had great APIs and libraries for Android development. PubNub would be more useful for future development as well, as it has capability for push notifications for mobile applications.

Arduino development has proven to be a fun learning experience for me in my classes that require it. I was happy to work with an Arduino board for this project, however doing so caused most of my problem. First, with the many parts of the device's hardware necessary for functionality, programmable memory was an issue for the board. I had to order the more capable Mega 2560 board for development. Also, after deciding to replace the Arduino WiFi shield for a more cost effective TI CC3000 based WiFi module, rewriting of code had to be done as well as replacing code used in necessary libraries that utilized Arduino's WiFi library.

All in all, this was an amazing learning experience that grew my knowledge in many aspects of software and hardware development. It will be easy to use what I learned these last two quarters on future projects. Going through the endless struggles and tough road blocks made me a better engineer as well as a better project manager. The tools gained through this process are invaluable.