Remote Home

System Design Document | Current Version [1.0.0]

Prepared By:

Christopher Jensen

Joshua Kinkade

Brian Vogel

James Wiegand

Revision History

Date	Author	Version	Comments
12/7/12	James Wiegand	0.0.0	Rough Draft
12/15/12	Joshua Kinkade	0.0.6	Rewrote using template and added detail
12/16/12	Christopher Jensen	0.0.7	Made some style corrections and added
5/7/13	Christopher Jensen	0.0.8	Updated elements, did final review for grammar and spelling, and added some images and details to hardware setup.

Table of Contents

1 Overview

The purpose of this document is to give the reader an understanding of iPhone Home Remote. iPhone Home Remote is a system that will allow users to control devices in their house from anywhere in the world, using their iPhone. Although the system contains implementations for garage doors, the iPhone Home Remote system is easily extendable to allow any electronic device to be controlled with only minor changes to the system. The iPhone Home Remote system has four major components:

i. An iPhone application, which acts as a remote control,

ii. A Base Station in the house, which connects the devices to the internet,

iii. A Resolution Server, which allows the iPhone application to connect directly to a Base Station, and

iv. The actual devices to be controlled.

1.1 Scope

This document describes what the purpose and requirements of our system are and how it works, including details about the major components and what technologies are used to implement the system. It also discusses our team and development process. It does NOT act as a user guide, nor does it describe in full a standard or metric for evaluation of technologies.

1.2 Purpose

The purpose of this project is to allow people to control electronic devices in their house over the internet using their iOS-compatible devices, and to create a formal protocol for passing data to such devices over TCP/IP.

1.2.1 iPhone Application

The users will mostly interact with the system using an application on their smart phone. This application allows them to register a Base Station with their phone, view all of their Base Stations, view and control all of the devices associated with each Base Station.

1.2.2 Base Station

The Base Station can connect all of its attached devices to the internet. It will register its IP address and unique identifier with the Resolution Server so the iPhone application will be able to connect directly to the Base Station. The Base Station will send commands from the iPhone to the appropriate device, and act as a basic authentication server to prevent unauthorized access. Configuration of the server can be done through a remotely-accessible web application running on an Apache server on the base station. It is also possible to manually configure the base station by editing the devices.ini and users.ini files with a text editor while locally logged into the base station.

1.2.3 Resolution Server

The Resolution Server will store the unique identifiers and IP addresses of all Base Stations that have been setup, will handle requests from the iPhone application to get the IP address of a particular Base Station, and will receive updated IP addresses from all of the Base Stations.

1.2.4 Devices

The devices to be controlled are electronic devices that can either receive commands from the Base Station and reply, or devices which can be controlled by such a device externally. A device could be anything from a lamp, which can be operated by opening or closing a switch, to a complex programmable thermostat that would allow users to adjust the temperature or create heating schedules from their iPhone. The present implementation includes a garage door opener as a sample device and example of how to implement the protocol that the system provides.

1.3 System Goals

The goal of the iPhone Home Remote system is to provide a complete, end-to-end solution to allow people to control devices in their house from anywhere they can connect to the internet.

1.4 System Overview and Diagram

The four major parts of the Remote Home system are the iPhone application, the Base Station, the Resolution Server and the actual devices that are controlled.

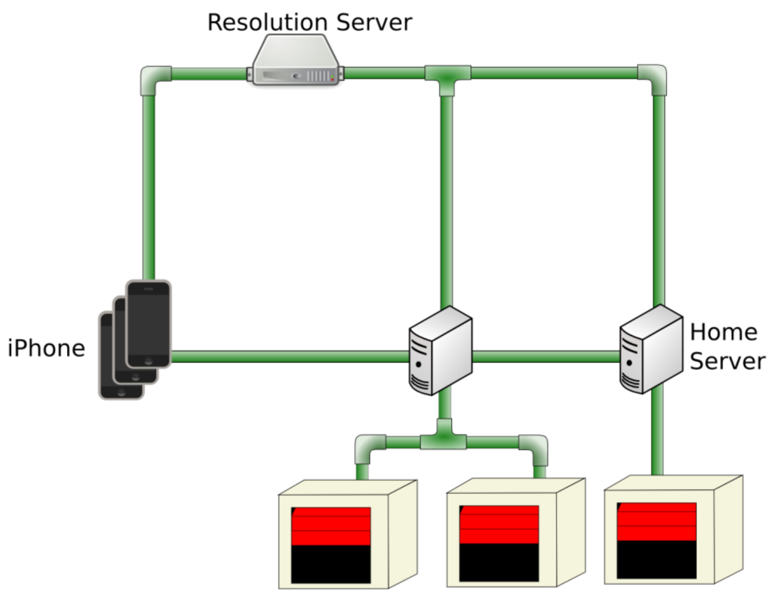


Figure 1: System Architecture

1.4.1 iPhone Application

The iPhone application allows users to view or add Base Stations and view and control devices associated with those Base Stations. It will use serial numbers on each of the Base Stations associated to get an IP address from the Resolution Server and will use this data to connect directly with a Base Station using TCP. Communications are currently sent “In the clear”, although with some modification it would be possible to use TLS or other encryption technology. The application stores the serial number, password, and user-defined friendly name of each base station on the phone.

1.4.2 Base Station

The Base Station will handle connections from the iPhone and send commands to the appropriate device using the device’s user-defined friendly name. It will also send feedback from the devices to the iPhone, and periodically update the Resolution Server with its global IP address. At present, the Base Station is only designed to handle communications over TCP/IP or Serial to communicate with devices, but it would not be hard to expand to other communication protocols in Python, such as Bluetooth/HCI connections.

1.4.3 Resolution Server

The Resolution Server stores the serial numbers and associated IP addresses of various base stations and handles requests from the iPhone for the IP address associated with a given serial number. It also handles requests from the Base Station to update its IP address.

1.4.4 Devices

The actual devices that will be controlled by the iPhone are regular electronic devices currently available on the market, or devices specially modified to be controlled by a microcontroller. The devices will be capable of receiving commands from the Base Station and sending information that needs to be sent from the iPhone to the Base Station.

1.4.5 Technologies Overview

The system uses a variety of technologies. Each component’s technology is discussed in the component’s sub section in section 4. All communication except for communications from base station to device shall be based on TCP/IP and use JSON to structure information sent between components. Communications between devices and base station will be a serial format which will include methods to query data based upon an index system.

2 Project Overview

2.1 Team Members and Roles

James Wiegand is our team leader and developed the core of the iPhone application. Joshua Kinkade developed the individual Device controllers for the iPhone application and the Resolution Server. Christopher Jensen worked on the hardware and the Base Station software. Brian Vogel joined our team in January and worked with Christopher on the hardware.

2.2 Project Management Approach

Our project is using the Agile development process. Our sprints are mostly three weeks long. We are using Trello to keep track of our backlog and github to host our code repository.

2.3 Phase Overview

In the first phase of our project we focused on getting the system operational. To limit the amount of work in this, we focused on getting one device, the garage door, working. This allowed us to build each component of our system and ensure that they work together.

2.4 Terminology and Acronyms

Base Station – A lightweight server that will be in the users house. This device controls and manages devices and communicates directly with the iPhone app.

Cocoa Touch – Apple's framework for iOS applications.

Device – A physical object that the user wishes to control with their iOS application. Examples would be garage doors, and sprinkler systems. The only device created by this team is a garage door opener.

iOS – The operating system that is present on iPhones and iPads. In this context iOS refers to the iOS 6.0

Resolution Server – A server operated by our client. The server has a database that will store associations between a serial numbers and IP address.

Serial Number – The MAC address of a base station's primary network card, as reported by python's uuid library.

TCP – Transmission Control Protocol – protocol that manages the transfer of data from one computer to another.

UIAlertView – The standard iOS dialog box for alerting the user with important information.

2.5 Requirements

2.5.1 Smart Phone App

The user should be able to control from anywhere in the world with internet access. The client gave specific requirements for garage doors and sprinkler systems. Our system is able to handle garage doors, and can be expanded to any other device that is designed for our system. For the garage door, the application should be able to tell if the door is open or closed or in between and control it with a button. Our garage door control handles the status display by showing a simple garage door graphic.

2.5.1.1 User Stories

1. When the user first starts up the app, or when a user decides to add a device, the add device screen will load. This screen will have two text input boxes. The first box will have a field for the device identification (DID). The second box will have a field for the device name. The DID will be located on each device, the name is a name the user specifies for her/his own use. (JW)

2. When the user opens the app and there is data in the SQLite database, the user will be presented with a list of devices. Each device will be under the heading of its appropriate type. For example if the user has a garage door opener called "Right Garage Door Opener" it would be placed under the "Garage Door" header. Each cell will have a green or red dot to indicate the status of that device (green is online, red is not). The user will have the option to add or delete devices. If a user touches a device that is online it will open the appropriate controller view. If the user touches an offline object she/he will be presented with a screen that gives an error message. (JW)

3. The status view will appear when the user clicks on a device from the main view that is listed as offline, or when the user selects status from the navigation bar on a controller view. The status view will tell users the status of the device (online, offline) at the top of the view. If the device is offline a code will be presented in the middle of the view, this code will be for technical use. For example if the user has a garage door opener called "Right Garage Door Opener" that is listed offline, because the server cannot communicate with the device the code ERR:001 could be displayed. At the bottom of the view there will be a text box that has an English description of the problem and a potential remedy to said problem. In the above example "The base station cannot communicate with the device, please ensure that the device is turned on and connected to the base station." could be displayed. (JW)

4. The garage door controller view will be loaded when a user clicks an online garage door type from the main view. This view, like all Controller Views will have a navigation bar with elements to go back to the main view or go to the status view. The view will have a percentage bar that will update the device on the status of the door. It will show either that the door is open, closed, or somewhere in between. It will also have a dynamic button to toggle the door. The text of the button will be based on the state. The view will get the initial state of the door from the device when it first loads. (JW, JK)

5. When the user pushes the toggle button, the view will tell the device to open the garage door and disable the button. Once that request has been completed, it will adjust the garage door picture to the correct state. If the door is opening, then it will re-enable the button

immediately. If the door is closing, then the controller will confirm with the device that the door closed before re-enabling the button. If the door failed to close, then the controller will open the garage door picture half way. (JK)

6. If the garage door controller view cannot connect to the server at any time, it will alert the user to the problem and offer to either cancel or retry. If the user wants to cancel, the app will return to the devices list. If the user wants to retry, the controller will resend the request. (JK)

2.5.2 Hardware Control

The client suggested that we use a Raspberry Pi, BeagleBone, or an Arduino to control the devices. We decided to use Arduinos because Raspberry Pi is more complicated than what is needed for our purposes and they are more readily available than Raspberry Pis are right now. Each of our devices will be controlled by an Arduino and will connect to a central Base Station.

2.5.3 Base Station

Because we generalized what devices our system would control, we added a base station that will sit in the user's house and coordinate communication between the application and all of the devices in the house.

2.5.3.1 Base Station User Stories

1. The user purchases a new hardware device and wants to add it to the system. The user points a web browser at the Base Station, clicks on the Devices menu, then clicks on the add device button. The user fills in the appropriate fields and the server reconfigures itself on the next call from a device.
2. The user wants to give permission to a neighbor to open the garage to feed the cats. The user accesses the users menu on the base station and adds a user in a group which is only associated with the garage door that the neighbor should be able to open. The neighbor can now open the door, but cannot access other devices in the system.

2.5.4 Communication

The client suggested that we use a custom TCP/IP protocol for communication between our devices. We are creating a protocol that uses a direct two way connection between the smartphone and the Base Station and also between the smartphone or base station and the Resolution Server. It uses JSON to send requests. The protocol is detailed in Appendix II.1.

2.5.5 Resolution Server

Because the Base Stations will be located at the users' houses, they will most likely have dynamically assigned IP addresses, we are creating a server that will associate a Base Station's unique serial number with its current IP address. The Base Stations will periodically send updated IP addresses to the Resolution server. When the phone application wants to connect to a Base Station, it will first send the serial number of the Base Station to the Resolution server to get the Station's IP address. This will allow the smart phone to establish a direct connection to the Base Station and keeps our system from depending on outside providers for service, since anyone can purchase static storage space on the internet today.

3 Design and Implementation

This section is used to describe the design details for each of the major components in the system. This section is not brief and requires the necessary detail that can be used by the reader to truly understand the architecture and implementation details without having to dig into the code.

3.1 iPhone Application

3.1.1 Technologies Used

The iPhone application will be a native iOS application developed using the Cocoa Touch application framework.

3.1.2 Component Overview

The iOS application will act as the remote control for the Home Remote system. The application will consist of views. The application will also use a SQLite database that will store the base stations that have been registered. When the application is first started the "first time registration" controllers will run: this will require the user to register a valid base station. If the user has at least one base station registered, the application will start the "main view" controllers. This is a UINavigation view controller that will present a list of base stations. In addition, an add button will be in the upper right hand corner of the list view. If the user presses this button, they can add a new base station. The user can swipe across a cell of a base station to delete the base station from the SQLite database. If the user selects a base station, a new list will populate with the individual devices available on that base station. The user will have an edit button in the upper right hand corner of the list view. If the user presses this button they will be presented with a form where they can modify the properties of the base station. If a user selects a device, they will be presented with the correct device controller. (JW)

3.1.3 Phase Overview

i. User Interface for viewing Base Stations and their associated devices
ii. User Interface for registering a Base Station with the iPhone
iii. Getting IP addresses for Base Stations from the Resolution Server

iv. Device Specific view controllers

3.1.4 Data Flow Diagram

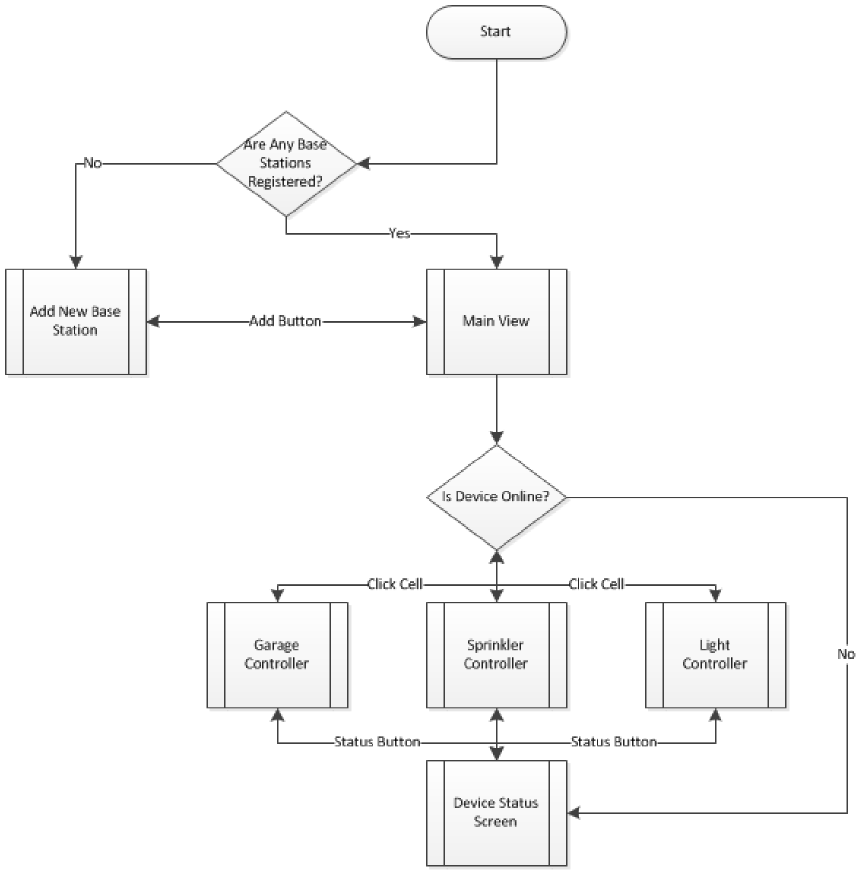


Figure 2: iOS application flow

3.1.5 Design Details

3.1.5.1 First time registration (iOS)

The purpose of the first time registration is to force the user to register a base station with the phone so that they can control devices. We will display this view if there are no base stations registered in the SQLite database.

3.1.5.2 Instruction View Controller

The purpose of this view controller is to display a scroll view with instruction embedded in it. These scroll view will instruct the user to set up their base station and connect devices. At the bottom of the view controller will be a button so that the user can advance to the registration view.

3.1.5.3 Registration View Controller

The purpose of this view controller is to allow the user to register a new base station. The view will consist of three text boxes and a register button. The view controller will check to see if all three fields are filled before connecting to the resolution server. If any of the fields are empty, a UIAlertView will be displayed telling the user to fill out the empty field.

If all three fields are populated, and the user clicks the register button, the device will attempt to make a TCP connection to the resolution server on port 8128. At this point, the device will start a timeout timer. If the TCP connection fails to open before the timeout fires, the system will close the connection and present a UIAlertView to the user. The alert will instruct the user to check their connection and/or try again later.

If the connection is successful, the server will send the connection DDNSConnected (See "Bidirectional iOS to Resolution Server Communication") signal to the phone. At this point, the phone will send the HRHomeStationsRequest with the serial number provided by the serial number field, which causes the Resolution Server to look up the serial number. If the Resolution Server finds the serial number, it will respond with a HRHomeStationReply containing the correct IP address and serial number. If the Resolution Server fails to find the serial number it will respond with HRHomeStationReply containing 'null' for the IP address and the correct serial number.

If the phone receives a null for the IP address it will present the user with an UIAlertView. This view will inform the user to check the serial number and make sure that they set up the base station correctly. If an IP address is sent, the device will register the device in the SQLite server and present a UIAlertView informing the user that the device was successfully registered. After dismissing this message, the device will go into the main view.

3.1.5.4 Garage Door View Controller

This view controller will allow a user to control a single garage door. The user interface consists of a simple animated garage door graphic at the top of the screen and a large button at the bottom. The user can open and close the door by pressing the button. The label will change to fit the current action. It will say 'open' when the door is closed and 'close' when the door is open. In addition, the user can swipe the garage door picture up to open the door and down to close it. If there is an object preventing the door from closing, the garage door graphic will move to halfway closed. Once the object has been removed the user can finish closing the door. The view does not have any way to directly tell that there is, in fact, something in the door at any given time.

3.2 Base Station (CJ)

3.2.1 Technologies Used

The base station leverages the same Python interface used by the Resolution Server, as well as the PySerial interface for sending serial data. The sample uses PySerial to communicate via USB with the individual device.

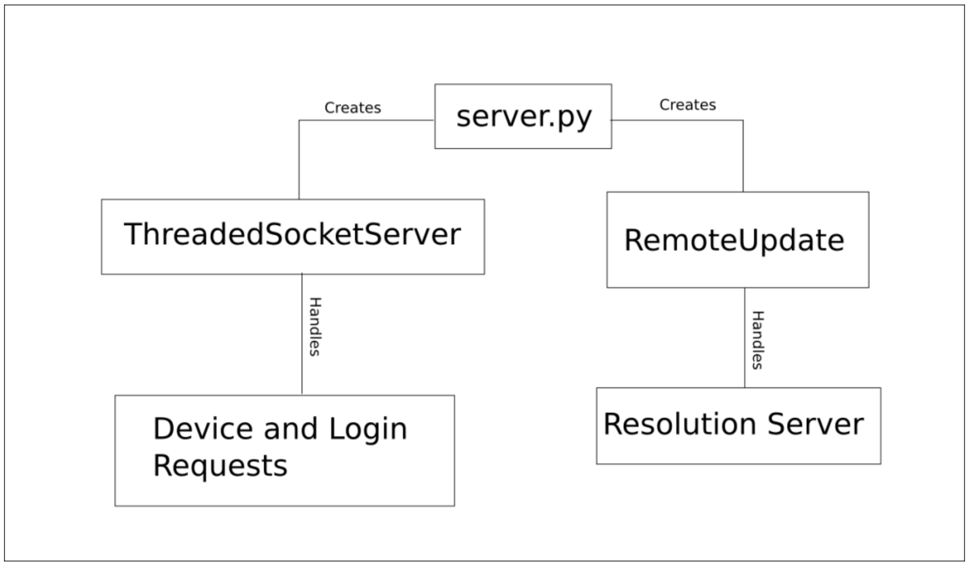
3.2.2 Component Overview

The Base Station will run on an Intel x86 or amd64 architecture CPU, with at least 1 MB RAM and 50KB of disk space. The base station can be deployed to any system which supports Python and PySerial, so theoretically other CPU architectures could be targeted if Python and PySerial are available.

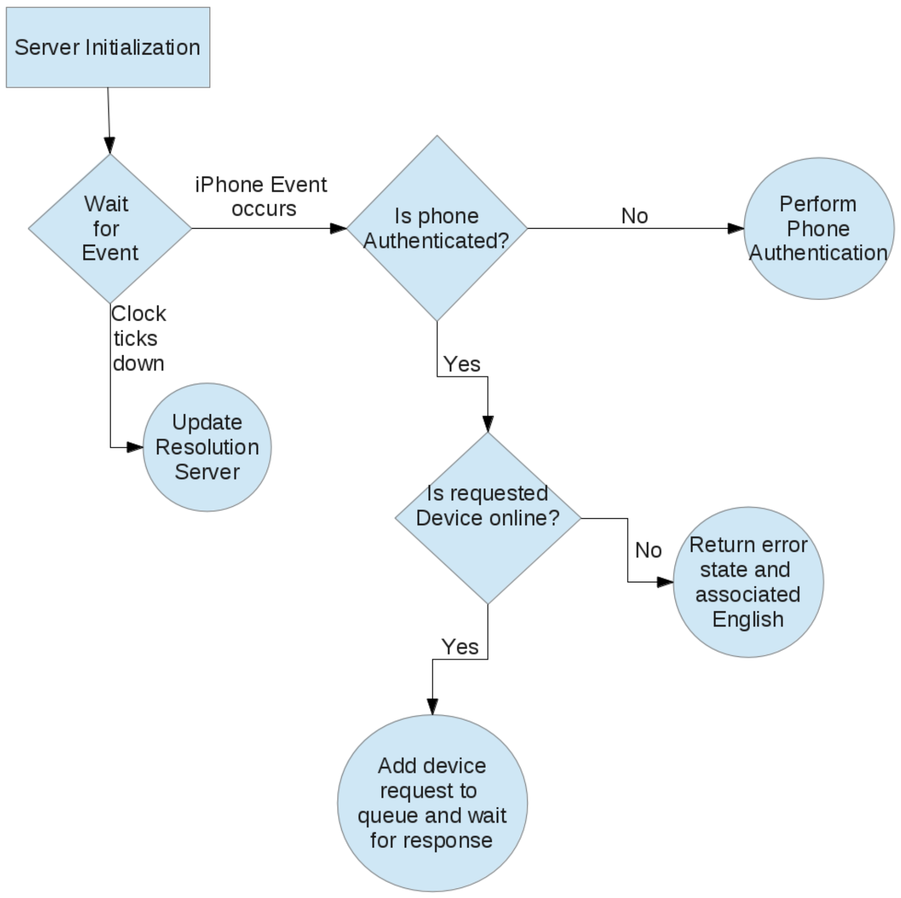
3.2.3 Phase Overview

1. Create an executable to handle basic network communications with the iPhone app: Authentication, General Receive, General Send
2. Create or expand an executable to communicate with devices: General Receive, General Send, ACK, NACK
3. Have the ability to populate a device pool and credentials from the base station
4. Get the web interface which integrates database management and basic registration tasks constructed.

3.2.4 Architecture Diagram



3.2.5 Data | Logic Flow Diagram


Server Initialization
Wait for Event
iPhone Event occurs
Is phone Authenticated?
No
Perform Phone Authentication
Clock ticks down
Update Resolution Server
Yes
Is requested Device online?
No
Return error state and associated English
Yes
Add device request to queue and wait for response

3.2.6 Design Details

3.2.6.1 *Server Details*

Presently, the base station works using the Python built-in SocketServer in threaded mode to handle requests. It also starts a daemon thread that occasionally sends a data parcel to the resolution server to keep the system updated. The serial number is generated by the python uuid library, which queries the MAC address of whichever network card is presently active. This allows for a reasonably unique identifier which SHOULD not change.

3.2.6.2 *Configuration Files*

The different devices are stored in an associated ini file. ini format was selected for its simplicity and small size; XML or similar verbose systems may not fit easily on a small footprint device and require additional effort to parse. Each entry in the file contains a key of the device name, an associated group, Device ID, and the interface on which the device should be contacted. It is theoretically possible to communicate with other devices on port 8128 by specifying an IPv4 address (xx.xx.xx.xx) as the interface, otherwise a serial device is assumed. The server will attempt to open said interface using the PySerial library.

Users have a separate ini file which contains the user's password (unhashed) and group. The special group “All” (case-insensitive) is considered an alias for all devices attached to the station, and can be considered an administrative group. Beyond that, any user's group is matched against the group of each device before a response is sent from the base station, so any client apps will only be aware of devices the user is permitted to access.

3.3 Resolution Server (JK)

3.3.1 Technologies Used

The Resolution Server is written in Python 2.7. It uses the SQLite database to store data and TCP to communicate.

3.3.2 Component Overview

The Resolution Server stores the IP address of each Base Station, along with a unique identifier for that station, and allows the iOS application to ask for the IP address of a Base Station. The server will run as a background daemon on a computer with a domain name or a static IP address. It will use port number 8128. The IP address and identifier of the Base Stations are stored in a SQLite database. The server itself will be written in Python 2.7. It will be written in a few classes. The Finder classes will handle the SQLite database. The Server class will handle connections.

3.3.3 Phase Overview

i. Make the server correctly respond to requests for IP addresses from the iOS application.
ii. Make the server correctly respond to requests to update an IP address from a Base Station.
iii. Make the server run as a daemon, so the computer running the program doesn’t need a terminal open all of the time.

3.3.4 Architecture Diagram

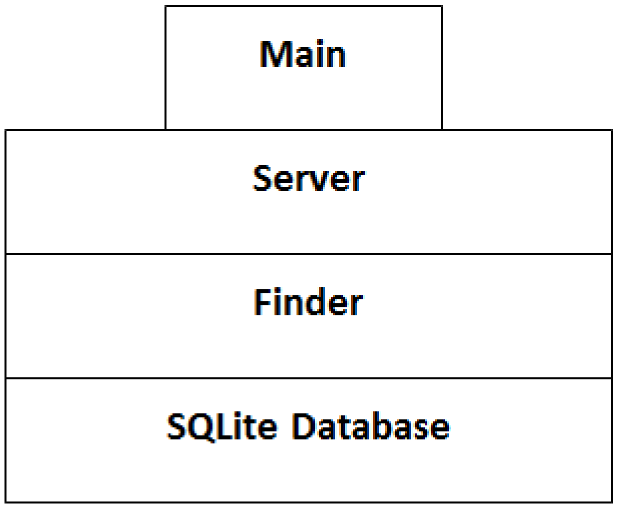


Figure 3: Resolution server architecture

3.3.5 Design Details

3.3.5.1 SQLite Database

This database has a single table, called devices, with two columns: ID and IP. Both columns are text and the column ID is the primary key.

3.3.5.2 Finder Class

The Finder class abstracts database interaction with the rest of the program. It will have methods that roughly correspond to the possible requests made to the server. It may also have some utility methods.

3.3.5.3 Server Class

The Server Class handles connections with the client. It waits for a connection and, when it gets one, immediately responds using the DDNSConnected protocol described in Appendix II.1.1.1. This is required for the iPhone application to connect properly. It will then wait for the actual request from the client. Once it has the request, it will call the appropriate method of the Finder class to retrieve or store information.

3.3.5.4 Main.py

The main Python file contains the function main. The main function is just the entry point for the program. It instantiates a Server Object and calls its main loop. Once the loop returns, the program exits. This function will also be responsible for converting the process into a daemon.

3.4 Devices

3.4.1 Technologies Used

The garage door controller is an Arduino microcontroller using a relay shield and a prototype shield. The relay shield is used to close the activation circuit on the garage door opener. This simulates the wall mounted door controller being pressed. The prototype shield is used to gain access to pins covered by the relay shield. The pins accessed through the relay shield are connected to sensors that detect when the garage door is opened, closed, or somewhere in between.

3.4.2 Component Overview

The relays close and open the garage door's "wall button" circuit to operate a standard garage door opener. The device listens on its USB connection for a signal from the base station, then reports its status (or an error, if applicable). If it receives a signal ASCII 1, it returns an integer value indicating the state of the door. The possible values are

0	OPEN
1	OPENING
2	CLOSED
3	CLOSING
4	PARTIAL

If it receives a signal ASCII 0, it actuates relay 4 closed and open, causing the garage door to operate.

3.4.3 Phase Overview

i. Explore how the garage door operates.

ii. Acquire hardware for controlling the garage door.

iii. Set up hardware to operate as specified.

iv. Integrate communication with base station

3.4.4 Architecture Diagram

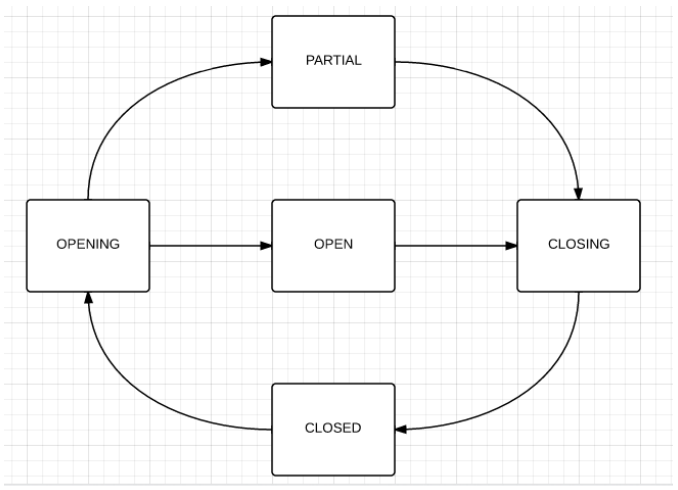
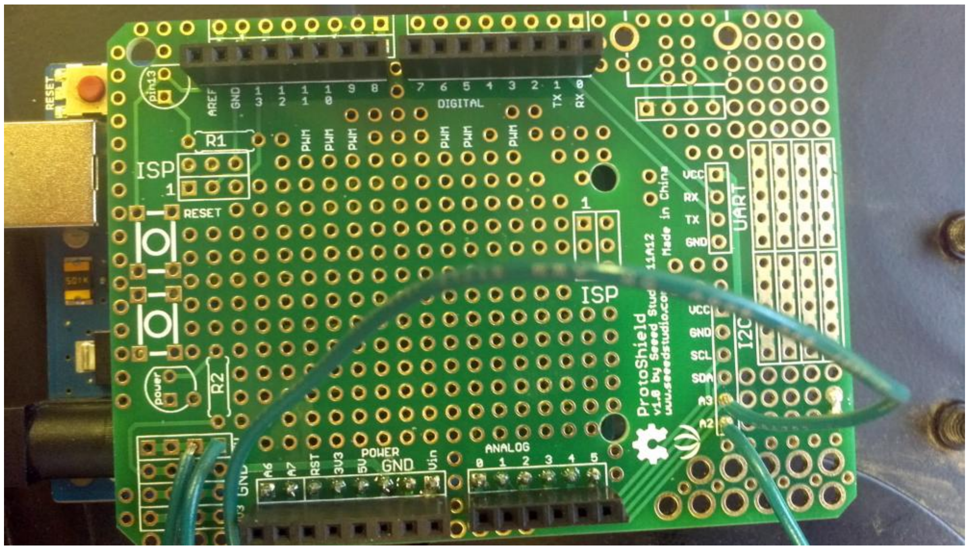


Figure 4: Garage Door State Diagram

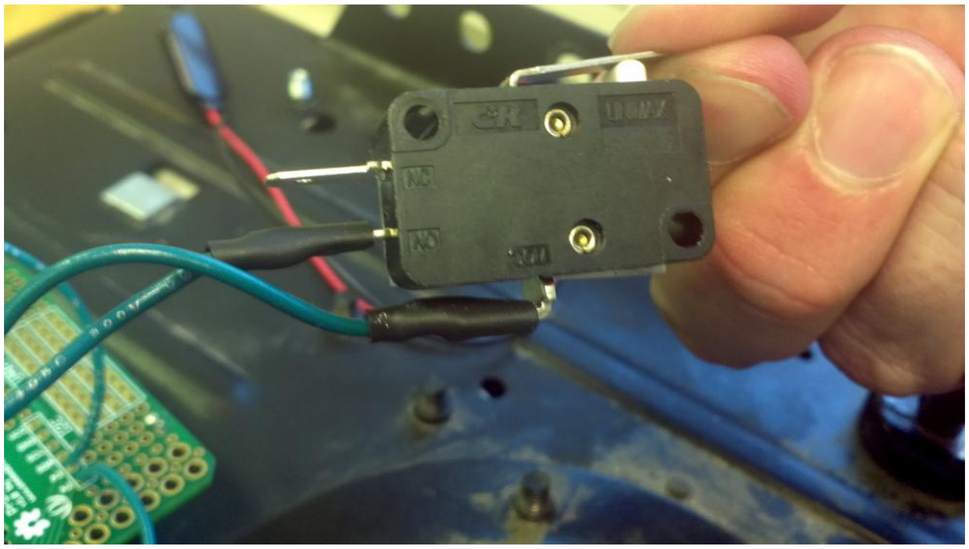
3.4.5 Design Details

3.4.5.1 *Hardware Design (CJ)*

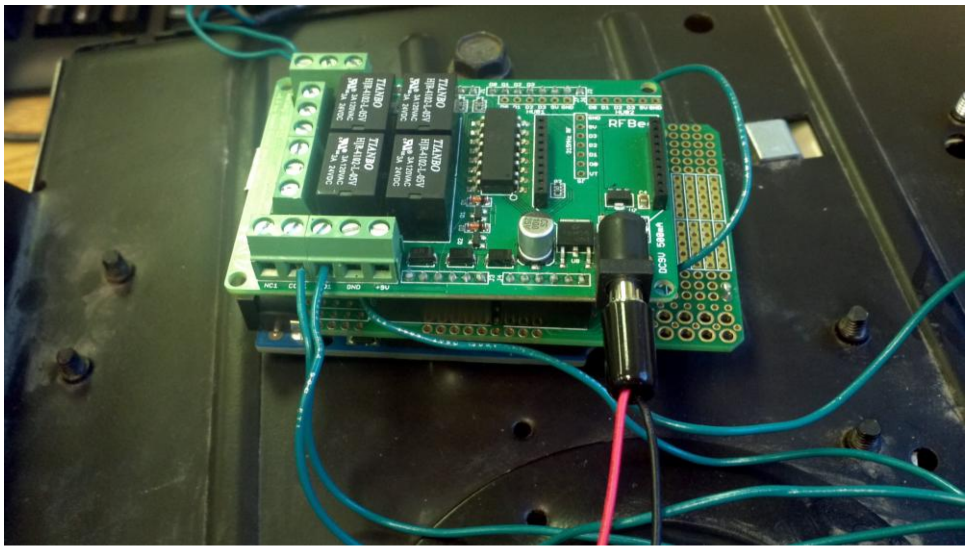
The garage door controller was built on an Arduino Uno with a Seeed Studio Protoshield and Seeed Studio Relay Shield. The Protoshield sits between the Uno and the Relay shield, and is soldered with the pins offset from the sockets, allowing the Relay shield to stack on top.



The Protoshield is used to provide the switches used to mount on the gantry system. The leads should attach to the +5V line on one end and either A2 or A3 on the opposite end. This allows a switch to provide +5V to the analogue inputs.



The input should be wired to COM (Common), with NO (Normally Open) wired to the +5V and NC (Normally Closed) optionally wired to GND (Board's common ground). The optional wire can prevent the floating wire from accidentally providing false positives.



On top of the Protoshield, mount the Relay Shield. Wire the COM1 and NO1 to the garage door opener's slots for a wall button. The order will not matter; the relay effectively acts as a simple switch controlled by the Arduino board. Finally, you will need a 9V power supply. For the prototype, we took a Size D plug and attached it to a 9V battery snap, allowing us to use 9V batteries to operate the device. The device can be powered by a single 9V battery for about 20 hours before replacement is needed, so this is not the recommended method for getting 9V to the board long-term. Remember, the Arduino is not meant to handle more than 12V of input potential. 9V is highly recommended, and is the minimum required for the relay shield to operate.

3.4.5.2 Garage Door Controller

The Garage Door Controller file is the Ardunio code that controls the garage door. When a device connects to the garage door controller, it will check the sensors for the current state of the garage door. It will toggle the voltage on the relay shield, activating the garage door, then change the current state of the garage door according to the state diagram in Figure 4. It will also return the current state of the garage door when queried.

4 System and Unit Testing

4.1 iPhone Application

4.1.1 RHBaseAddBaseStation View Tests (JW)

Fill nothing – Wireless Off	
Description:	Have the application in the first run screen. Attempt to press the register button with no fields filled.
Expected Result:	CMPopup should appear informing user to fill the serial number field.
Actual Result:	CMPopup should appear informing user to fill the serial number field.
Pass/Fail:	Pass

Fill Serial Number– Wireless On	
Description:	Have the application in the first run screen. Attempt to press the register button with only the serial number field filled.
Expected Result:	CMPopup should appear informing user to fill the name field.
Actual Result:	CMPopup should appear informing user to fill the name field.
Pass/Fail:	Pass

Fill Serial Number– Wireless Off	
Description:	Have the application in the first run screen. Attempt to press the register button with only the serial number field filled.
Expected Result:	CMPopup should appear informing user to fill the name field.
Actual Result:	CMPopup should appear informing user to fill the name field.
Pass/Fail:	Pass

Fill Name – Wireless On	
Description:	Have the application in the first run screen. Attempt to press the register button with only the name field filled.

Expected Result:	CMPopup should appear informing user to fill the serial number field.
Actual Result:	CMPopup should appear informing user to fill the serial number field.
Pass/Fail:	Pass

Fill Name – Wireless Off	
Description:	Have the application in the first run screen. Attempt to press the register button with only the name field filled.
Expected Result:	CMPopup should appear informing user to fill the serial number field.
Actual Result:	CMPopup should appear informing user to fill the serial number field.
Pass/Fail:	Pass

Fill Password – Wireless On	
Description:	Have the application in the first run screen. Attempt to press the register button with only the password field filled.
Expected Result:	CMPopup should appear informing user to fill the serial number field.
Actual Result:	CMPopup should appear informing user to fill the serial number field.
Pass/Fail:	Pass

Fill Password – Wireless Off	
Description:	Have the application in the first run screen. Attempt to press the register button with only the password field filled.
Expected Result:	CMPopup should appear informing user to fill the serial number field.
Actual Result:	CMPopup should appear informing user to fill the serial number field.
Pass/Fail:	Pass

Fill All – Wireless Off	
Description:	Have the application in the first run screen. Attempt to press the register button with all fields filled correctly.
Expected Result:	A dialog should appear informing the user to check their connection and try again later.
Actual Result:	A dialog should appear informing the user to check their connection and try again later.
Pass/Fail:	Pass

Bad Password – Wireless On	
Description:	Have the application in the first run screen. Attempt to press the register button with all fields filled however the password is bad.

Expected Result:	A dialog should appear informing the user to check their password.
Actual Result:	A dialog should appear informing the user to check their password.
Pass/Fail:	Pass
Bad Serial Number – Wireless On	
Description:	Have the application in the first run screen. Attempt to press the register button with all fields filled however the serial number is bad.
Expected Result:	A dialog should appear informing the user to check their serial number.
Actual Result:	A dialog should appear informing the user to check their serial number.
Pass/Fail:	Pass

Fill All – Wireless On – No DDNS Server	
Description:	Have the application in the first run screen. Attempt to press the register button with all fields correct however there is no DDNS server.
Expected Result:	A dialog should appear informing the user to check their connection and try again later.
Actual Result:	A dialog should appear informing the user to check their connection and try again later.
Pass/Fail:	Pass

Fill All – Wireless On	
Description:	Have the application in the first run screen. Attempt to press the register button with all fields correct.
Expected Result:	A dialog should appear informing the user of success and the device list should load.
Actual Result:	A dialog should appear informing the user of success and the device list should load.
Pass/Fail:	Pass

4.1.2 RHBaseStationTableViewController View Test (JW)

Connect to Base Station – Bad IP – No DDNS	
Description:	Click on a base station in the list. Attempt to connect without the DDNS server to resolve the serial number.
Expected Result:	A dialog should appear informing the user to check their connection and try again later.
Actual Result:	A dialog should appear informing the user to check their connection and try again later.
Pass/Fail:	Pass

Connect to Base Station – No Base Station	
Description:	Attempt to connect to a base station that is offline.
Expected	A dialog should appear informing the user to make sure their base station is on

Result:	and connected to the internet.
Actual Result:	A dialog should appear informing the user to make sure their base station is on and connected to the internet.
Pass/Fail:	Pass

Connect to Base Station – Bad Password	
Description:	Attempt to connect to a base station that is offline.
Expected Result:	A dialog should appear informing the user to check their password.
Actual Result:	A dialog should appear informing the user to check their password.
Pass/Fail:	Pass

Edit Base Station – Wireless On	
Description:	Attempt to edit a base station.
Expected Result:	The edit screen should load.
Actual Result:	The edit screen should load.
Pass/Fail:	Pass

Edit Base Station – Wireless Off	
Description:	Attempt to edit a base station.
Expected Result:	The edit screen should load.
Actual Result:	The edit screen should load.
Pass/Fail:	Pass

Delete Base Station – Wireless On	
Description:	Attempt to delete a base station.
Expected Result:	The base station should be deleted
Actual Result:	The base station should be deleted
Pass/Fail:	Pass

Delete Base Station – Wireless Off	
Description:	Attempt to delete a base station.
Expected Result:	The base station should be deleted
Actual Result:	The base station should be deleted
Pass/Fail:	Pass

4.1.3 RHUpdateBaseStationViewController (JW)

Fill Nothing – Wireless On

Description:	Delete both fields and click the update button.
Expected Result:	A CMPopUp should appear informing the user to fill the name field. If the user clicks base no changes should be made.
Actual Result:	A CMPopUp should appear informing the user to fill the name field. If the user clicks base no changes should be made.
Pass/Fail:	Pass

Fill Nothing – Wireless Off	
Description:	Delete both fields and click the update button.
Expected Result:	A CMPopUp should appear informing the user to fill the name field. If the user clicks base no changes should be made.
Actual Result:	A CMPopUp should appear informing the user to fill the name field. If the user clicks base no changes should be made.
Pass/Fail:	Pass

Fill Name – Wireless On	
Description:	Delete both fields, fill the name field and click the update button.
Expected Result:	A CMPopUp should appear informing the user to fill the password field. If the user clicks base no changes should be made.
Actual Result:	A CMPopUp should appear informing the user to fill the password field. If the user clicks base no changes should be made.
Pass/Fail:	Pass

Fill Name – Wireless Off	
Description:	Delete both fields, fill the name field and click the update button.
Expected Result:	A CMPopUp should appear informing the user to fill the password field. If the user clicks base no changes should be made.
Actual Result:	A CMPopUp should appear informing the user to fill the password field. If the user clicks base no changes should be made.
Pass/Fail:	Pass

Fill Password – Wireless On	
Description:	Delete both fields, fill the password field and click the update button.
Expected Result:	A CMPopUp should appear informing the user to fill the name field. If the user clicks base no changes should be made.
Actual Result:	A CMPopUp should appear informing the user to fill the name field. If the user clicks base no changes should be made.
Pass/Fail:	Pass

Fill Password – Wireless Off	
Description:	Delete both fields, fill the password field and click the update button.
Expected Result:	A CMPopUp should appear informing the user to fill the name field. If the user clicks base no changes should be made.

Actual Result:	A CMPopUp should appear informing the user to fill the name field. If the user clicks base no changes should be made.
Pass/Fail:	Pass

Fill All – Wireless On	
Description:	Delete both fields, fill the all fields and click the update button.
Expected Result:	A dialog should pop up informing the user that their changes have been saved. The changes should be saved in the database.
Actual Result:	A dialog should pop up informing the user that their changes have been saved. The changes should be saved in the database.
Pass/Fail:	Pass

Fill All – Wireless Off	
Description:	Delete both fields, fill the all fields and click the update button.
Expected Result:	A dialog should pop up informing the user that their changes have been saved. The changes should be saved in the database.
Actual Result:	A dialog should pop up informing the user that their changes have been saved. The changes should be saved in the database.
Pass/Fail:	Pass

4.1.4 RHDeviceViewController (JW)

Refresh – Wireless On	
Description:	Press the refresh button.
Expected Result:	Device states should refresh
Actual Result:	Device states should refresh
Pass/Fail:	Pass

Refresh – Wireless Off	
Description:	Press the refresh button.
Expected Result:	A dialog should inform the user that the base station could not be reached. All devices states should go offline.
Actual Result:	A dialog should inform the user that the base station could not be reached. All devices states should go offline.
Pass/Fail:	Pass

Select item that is offline – Wireless On	
Description:	Select a device that is offline.
Expected Result:	Should load the error screen

Actual Result:	Should load the error screen
Pass/Fail:	Pass

Select item that is offline – Wireless Off	
Description:	Select a device that is offline.
Expected Result:	Should load the error screen
Actual Result:	Should load the error screen
Pass/Fail:	Pass

Connect to device that is online – No Base Station	
Description:	Attempt to connect to a device that is online when the base station is offline.
Expected Result:	A dialog should inform the user that the base station could not be reached. All devices states should go offline.
Actual Result:	A dialog should inform the user that the base station could not be reached. All devices states should go offline.
Pass/Fail:	Pass

4.1.5 RHGarageDoorViewController (JK)

getRequestDictForAction:andHumanMessage: - toggle request	
Description:	This test ensures that the getRequestDictForAction:andHumanMessage: correctly creates a toggle request. Parameters: 0, “Toggle” Properties: self.deviceID = @”George”, self.password = @”password”
Expected Result:	Valid request dictionary with action: 0, humanMessage: Toggle, deviceID: George,password: password, Type:str

Actual Result:	{
	    HRDeviceRequest =
	{
	    Data = 0;
	    DeviceID = George;
	    HumanMessage = Toggle;
	    Password: password;
	    Type: str;
	};
	}

Pass/Fail:	Pass

getRequestDictForAction:andHumanMessage: - query request	
Description:	This test ensures that the getRequestDictForAction:andHumanMessage: correctly creates a query request. Parameters: 1, “Query”

	Properties: self.deviceID = @"George", self.password = @"password"
Expected Result:	Valid request dictionary with action: 1, humanMessage: Query, deviceID: George,password: password, Type:str

Actual Result:	{
	HRDeviceRequest =
	{
	Data = 1;
	DeviceID = George;
	HumanMessage = Query;
	Password: password;
	Type: str;
	};
	}

Pass/Fail:	Pass

getRequestDictForAction:andHumanMessage: - nil human request	
Description:	This test ensures that a nil human message doesn't break getRequestDictForAction:andHumanMessage:. Parameters: 1, nil Properties: self.deviceID = @"George", self.password = @"password"
Expected Result:	Valid request dictionary with action: 1, humanMessage: "(null)", deviceID: George,password: password, Type:str

Actual Result:	{
	HRDeviceRequest =
	{
	Data = 1;
	DeviceID = George;
	HumanMessage = "(null)";
	Password: password;
	Type: str;
	};
	}

Pass/Fail:	Pass

Test getValueForKey:FromRequest:	
Description:	This test ensures that the getValueForKey:FromRequest: correctly gets a key from a request. Parameters: "Data",

	{
	HRDeviceRequest =
	{
	Data = 0;
	DeviceID = George;
	HumanMessage = Toggle;
	Password: password;
	Type: str;
	};
	}
	Properties: self.deviceID = @"George", self.password = @"password"

Expected Result:	1
Actual Result:	1
Pass/Fail:	Pass

Test getValueForKey:FromRequest:	

Description:	This test ensures that the getValueForKey:FromRequest: correctly handles a key that isn't in the request.
	Parameters: "NotInDict",
	{
	HRDeviceRequest =
	{
	Data = 1;
	DeviceID = George;
	HumanMessage = Query;
	Password: password;
	Type: str;
	};
	}
	Properties: self.deviceID = @"George", self.password = @"password"

Expected Result:	(null)
Actual Result:	(null)
Pass/Fail:	Pass

Test getValueForKey:FromRequest: - nil request	
Description:	This test ensures that the getValueForKey:FromRequest: correctly gets a key from a request. Parameters: "Data", nil

	Properties: self.deviceID = @”George”, self.password = @”password”
Expected Result:	(null)
Actual Result:	(null)
Pass/Fail:	Pass

toggleDoor	
Description:	This tests if toggleDoor correctly gets and sends a toggle request to sendRequest
Expected Result:	Should send toggle request to sendRequest
Actual Result:	Sends toggle request to sendRequest
Pass/Fail:	Pass

checkState	
Description:	This tests if checkstate correctly gets and sends a query request to sendRequest
Expected Result:	Should send query request to sendRequest
Actual Result:	Sends query request to sendRequest
Pass/Fail:	Pass

confirmClosed	
Description:	This tests if toggleDoor correctly gets and sends a toggle query to sendRequest
Expected Result:	Should send toggle query to sendRequest
Actual Result:	Sends toggle query to sendRequest
Pass/Fail:	Pass

sendRequest	
Description:	This tests if sendRequest correctly gets and sends a request
Expected Result:	Should send request
Actual Result:	Sends request
Pass/Fail:	Pass

toggleDoor	
Description:	This tests if toggleDoor correctly gets and sends a toggle query to sendRequest
Expected Result:	Should send toggle query to sendRequest

Actual Result:	Sends toggle query to sendRequest
Pass/Fail:	Pass

requestReturnedWithError	
Description:	This tests if requestReturnedWithError shows an alert view when it is called.
Expected Result:	Should show alert view
Actual Result:	Shows alert view
Pass/Fail:	Pass

Getting initial state	
Description:	Tests if the view controller gets the initial state of the garage door.
Expected Result:	Should update the garage door view to correct state and change the button label to the appropriate value
Actual Result:	Correctly updates the UI
Pass/Fail:	Pass

Normal door opening	
Description:	Tests if the view controller correctly sends the open request and updates the UI. Door must be closed to do this test.
Expected Result:	Should send a toggle request to the base station, update the garage door view to be open, and change the text on the button to “Close”
Actual Result:	Correctly sends the request updates the UI
Pass/Fail:	Pass

Normal door closing	
Description:	Tests if the view controller correctly sends the close request and updates the UI. Door must be open to do this test and must close successfully
Expected Result:	Should send a toggle request to the base station, update the garage door view to be closed, and change the text on the button to “Open”, and send a confirmation request. The door should stay closed.
Actual Result:	Correctly sends the requests and updates the UI
Pass/Fail:	Pass

Unsuccessful door closing	
Description:	Tests if the view controller correctly sends the open request and updates the UI. Door must be open to do this test and must fail to close
Expected	Should send a toggle request to the base station, update the garage door view

Result:	to be closed, and change the text on the button to “Open”, and send a confirmation request. The door should open when it gets the response to the confirmation.
Actual Result:	Correctly sends the requests and updates the UI
Pass/Fail:	Pass

Network connection failure - cancel	
Description:	Tests if the view controller correctly handles losing the connection to the base station and pops itself if the user cancels request.
Expected Result:	Should display an alert view telling the user that it can’t communicate with base station and offer to cancel request or retry.
Actual Result:	Correctly displays the alert and pops the view controller
Pass/Fail:	Pass

Network connection failure - retry	
Description:	Tests if the view controller correctly handles losing the connection to the base station and retries the request when user selects retry.
Expected Result:	Should display an alert view telling the user that it can’t communicate with base station and offer to cancel request or retry.
Actual Result:	Correctly displays the alert and retries the request.
Pass/Fail:	Pass

Swipe open normally	
Description:	Tests if the view controller correctly sends the open request and updates the UI when the user swipes up on the garage door picture. Door must be closed to do this test.
Expected Result:	Should send a toggle request to the base station, update the garage door view to be closed, and change the text on the button to “Close”.
Actual Result:	Doesn’t seem to register swipe
Pass/Fail:	fail

Swipe closed normally	
Description:	Tests if the view controller correctly sends the close request and updates the UI when the user swipes down on the garage door picture. Door must be closed to do this test.
Expected Result:	Should send a toggle request to the base station, update the garage door view to be closed, and change the text on the button to “Open”, and send a confirmation request. The door should stay closed.
Actual Result:	Doesn’t seem to register swipe
Pass/Fail:	fail

4.2 Base Station

server.py

Bad JSON string	
Description:	Send a bunch of nonsense to the server
Expected Result:	NAK: Invalid JSON is replied
Actual Result:	Replies NAK: Invalid JSON
Pass/Fail:	Pass

Bad Password string	
Description:	Send a valid HRLogin with an invalid pasword
Expected Result:	RHLoginSuccess:false
Actual Result:	Replies RHLoginSuccess:false
Pass/Fail:	Pass

Empty JSON	
Description:	Send a valid, but empty, bit of JSON
Expected Result:	NAK: Not a valid request.
Actual Result:	Replies NAK: Not a valid request.
Pass/Fail:	Pass

User login	
Description:	Send a valid login request for a user without admin access.
Expected Result:	Login success with a list of attached devices.
Actual Result:	Replies with a login success and list of attached devices that the associated user has priviledges to.
Pass/Fail:	Pass

Admin login	
Description:	Log in as a user in group ALL
Expected Result:	Login success with a list of attached devices.
Actual Result:	Replies with a login success and list of all attached devices
Pass/Fail:	Pass

Empty login	
Description:	Log in as a user whose group contains no devices
Expected Result:	Login success with an empty list
Actual Result:	Replies with a login success and empty list
Pass/Fail:	Pass

Device Request Good Password	
Description:	Send a device request to a device with a password whose group matches the device.

Expected Result:	Return value from device
Actual Result:	Replies with a return value from the device.
Pass/Fail:	Pass

Device Request Bad Password	
Description:	Send a login request to a device with a password whose group does not match the device.
Expected Result:	NAK: Bad Password
Actual Result:	Replies with a NAK: Bad Password
Pass/Fail:	Pass

Device Request Unavailable Device	
Description:	Send a login request to a device with a password whose group matches the device, but the device is not available.
Expected Result:	NAK: Failed to access [interface] (Bad interface or not connected.)
Actual Result:	Replies with a NAK: Failed to access [interface] (Bad interface or not connected.)
Pass/Fail:	Pass

Device Request Missing Password	
Description:	Send a login request to a device without a password field present.
Expected Result:	NAK: Invalid request
Actual Result:	Replies with a NAK: Invalid Request
Pass/Fail:	Pass

4.3 Resolution Server

4.3.1 Server.py

Valid HRHomeStationsRequest for single existing device	
Description:	This tests that the resolution server will correctly respond to a request for a single device that is in the database. The request sent to the server was: {“HRHomeStationsRequest”:[{“StationDID”:”mac”}]}
Expected Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”mac”,”StationIP”:”10.250.1.128”}]}
Actual Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”mac”,”StationIP”:”10.250.1.128”}]}
Pass/Fail:	Pass

Valid HRHomeStationsRequest for single non-existing device	
Description:	This tests that the resolution server will correctly respond to a request for a single device that is not in the database. The request sent to the server was: {“HRHomeStationsRequest”:[{“StationDID”:”imaginary”}]}
Expected Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”imaginary”,”StationIP”:null}]}
Actual Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”imaginary”,”StationIP”:null}]}
Pass/Fail:	Pass

Valid HRHomeStationsRequest for multiple existing devices	
Description:	This tests that the resolution server will correctly respond to a request for multiple devices that are in the database. The request sent to the server was: {“HRHomeStationsRequest”:[{“StationDID”:”mac”},{“StationDID”:”hello”}]}
Expected Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”mac”,”StationIP”:”10.250.1.128”},{“StationDID”:”hello”,”StationIP”:”10.250.1.128”}]}
Actual Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”mac”,”StationIP”:”10.250.1.128”},{“StationDID”:”hello”,”StationIP”:”10.250.1.128”}]}
Pass/Fail:	Pass

Valid HRHomeStationsRequest for existing and non-existing devices	
Description:	This tests that the resolution server will correctly respond to a request for multiple devicse that are not all in the database. The request sent to the server was:

	{“HRHomeStationsRequest”:[{“StationDID”:”mac”},{“StationDID”:”imaginary” }]}
Expected Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”mac”,”StationIP”:”10.250.1.128”}, {“StationDID”:”imaginary”,”StationIP”:null}]}
Actual Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”mac”,”StationIP”:”10.250.1.128”}, {“StationDID”:”imaginary”,”StationIP”:null}]}
Pass/Fail:	Pass

Valid HRHomeStationsRequest for multiple non-existing devices	
Description:	This tests that the resolution server will correctly respond to a request for multiple devices that are not in the database. The request sent to the server was: {“HRHomeStationsRequest”:[{“StationDID”:”imaginary”},{“StationDID”:”fake” }]}
Expected Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”imaginary”,”StationIP”:null}, {“StationDID”:”fake”,”StationIP”:null}]}
Actual Result:	{“HRHomeStationsReplyt”:[{“StationDID”:”mac”,”StationIP”:”10.250.1.128”}, {“StationDID”:”mac”,”StationIP”:”10.250.1.128”}]}
Pass/Fail:	Pass

Invalid JSON	
Description:	This test ensures that receiving bad JSON will not crash the server. The request sent to the server was: {“HRHomeStationsRequest”:{“StationDID”:”mac”}]}
Expected Result:	none
Actual Result:	none
Pass/Fail:	Pass

Invalid request type	
Description:	This test ensures that receiving an unrecognized request type will not crash the server. The request sent to the server was: {“HRImaginaryRequest”:{“StationDID”:”mac”}]}
Expected Result:	none
Actual Result:	none
Pass/Fail:	Pass

HRHomeStationsRequest with no devices given	
Description:	This test ensures that receiving an HRHomeStationsRequest with an empty device list will not crash the server. The request sent to the server was:

	{“HRHomeStationsRequest”:[]}
Expected Result:	none
Actual Result:	none
Pass/Fail:	Pass

HRHomeStationsRequest with no id given	
Description:	This test ensures that receiving an HRHomeStationsRequest without an id will not crash the server. The request sent to the server was: {“HRImaginaryRequest”:{“StationDID”:”}]}
Expected Result:	{“HRHomeStationsReplyt”:[{“StationDID”:””,”StationIP”:null}]}
Actual Result:	{“HRHomeStationsReplyt”:[{“StationDID”:””,”StationIP”:null}]}
Pass/Fail:	Pass

Valid HRHomeStationUpdate request	
Description:	This test ensures that the server will correctly respond to a valid HRHomeStationUpdate. The request sent to the server was: {“HRHomStationUpdate”:[{“StationDID”:”mac”}]}
Expected Result:	none
Actual Result:	none
Pass/Fail:	Pass

HRHomeStationUpdate request with no device given	
Description:	This test ensures that the server will correctly respond to an HRHomeStationUpdate without a device given. The request sent to the server was: {“HRHomStationUpdate”:[]}
Expected Result:	none
Actual Result:	none
Pass/Fail:	Pass

HRHomeStationUpdate request with no device id given	
Description:	This test ensures that the server will correctly respond to an HRHomeStationUpdate with an empty device id. The request sent to the server was:

	{"HRHomStationUpdate":[{"StationDID":"]}
Expected Result:	none
Actual Result:	none
Pass/Fail:	Pass

4.3.2 Finder.py

Finder.find with device in database	
Description:	This tests if the Finder class correctly retrieves the ip address for a device in the database. The argument passed to Finder.find was "mac"
Expected Result:	"10.250.1.128"
Actual Result:	"10.250.1.128"
Pass/Fail:	Pass
Finder.find with device not in database	
Description:	This tests if the Finder class correctly retrieves the ip address for a device not in the database. The argument passed to Finder.find was "imaginary"
Expected Result:	None
Actual Result:	None
Pass/Fail:	Pass

Finder.find with invalid device id	
Description:	This tests if the Finder class correctly retrieves the ip address for an invalid device id. The argument passed to Finder.find were None
Expected Result:	None
Actual Result:	None
Pass/Fail:	Pass

Finder.update with existing device id and up to date IP	
Description:	This tests if the Finder class correctly handles an update request with an existing device with an up to date IP address. The argument passed to Finder.find were "mac" and "10.250.1.128"

Expected Result:	Do nothing
Actual Result:	Do nothing
Pass/Fail:	Pass

Finder.update with existing device id and new IP	
Description:	This tests if the Finder class correctly handles an update request with an existing device with a new IP address. The argument passed to Finder.find were “hello” and “10.250.1.128”
Expected Result:	Update database with new IP address
Actual Result:	Updated database with new IP address
Pass/Fail:	Pass

Finder.update with existing new id and new IP	
Description:	This tests if the Finder class correctly handles an update request with a new device The argument passed to Finder.find were “aNewID” and “10.250.1.128”
Expected Result:	Update database with new device and IP address
Actual Result:	Updated database with new device and IP address
Pass/Fail:	Pass

4.4 Devices

Garage Door Toggle Close	
Description:	This will test the garage door going from an “Open” state to a “Closed” state. First make sure the garage door is in an “Open” state. This can be accomplished by interrupting the object sensors connected to the garage door. • Connect to the arduino using the Arduino Serial Monitor. • Enter “0” and then press “Send” in the Serial Monitor. • Wait ten (10) seconds. • Press the “Close” sensor. • Enter “1” and then press “Send” in the Serial Monitor.
Expected Result:	Current State 2.
Actual Result:	Current State 2.
Pass/Fail:	Pass

Garage Door Toggle Open	
Description:	This will test the garage door going from a “Closed” state to an “Open”. First make sure the garage door is in a “Closed” state. This has been tested by the previous unit test.

	• Enter “0” and then press “Send” in the Serial Monitor. • Wait ten (10) seconds. • Press the “Open” Sensor. • Enter “1” and then press “Send” in the Serial Monitor.
Expected Result:	Current State 0.
Actual Result:	Current State 0.
Pass/Fail:	Pass

Garage Door Closing State	
Description:	This will test the garage door going from an “Open” state, to a “Closing” state, and then to a “Closed” state. First make sure the garage door is in an open state. • Enter “0” and then press “Send” in the Serial Monitor. • Immediately enter “1” and then press “Send” in the Serial Monitor. • Wait ten (10) seconds. • Press the “Close” Sensor. • Enter “1” and then press “Send” in the Serial Monitor
Expected Result:	Current State 3, and then Current State 2.
Actual Result:	Current State 3, and then Current State 2.
Pass/Fail:	Pass

Garage Door Opening State	
Description:	This will test the garage door going from a “Closed” state, to an “Opening” state, and then to an “Open” state. First make sure the garage door is in a “Closed” state. • Enter “0” and then press “Send” in the Serial Monitor. • Immediately enter “1” and then press “Send” in the Serial Monitor. • Wait ten (10) seconds. • Press the “Open” Sensor. • Enter “1” and then press “Send” in the Serial Monitor.
Expected Result:	Current State 1, and then Current State 0.
Actual Result:	Current State 1, and then Current State 0.
Pass/Fail:	Pass

Garage Door Partial State	
Description:	This will test the garage door going from a “Closed” state, to a “Opening” state, to a “Partial” state. First make sure the garage door is in a “Closed” state. • Enter “0” and then press “Send” in the Serial Monitor. • Within ten (10) seconds, enter “0” and then press “Send” in the Serial

	Monitor. • Enter "1" and then press "Send" in the Serial Monitor.
Expected Result:	Current State 4.
Actual Result:	Current State 4.
Pass/Fail:	Pass

Garage Door Partial State to Closed State	
Description:	This will test the garage door going from a "Partial" state, to a "Closing" state, to a "Closed" state. First make sure the garage door is in a "Partial" state. • Enter "0" and then press "Send" in the Serial Monitor. • Immediately enter "1" and then press "Send" in the Serial Monitor. • Wait ten (10) seconds. • Press the "Close" sensor. • Enter "1" and then "Send" in the Serial Monitor.
Expected Result:	Current State 3, and then Current State 2.
Actual Result:	Current State 3, and then Current State 2.
Pass/Fail:	Pass

Garage Door Opening with Interrupt	
Description:	This will test the garage door going from a "Closed" state, to an "Opening" state, to an "Open" state, with an object placed in the door. First make sure the garage door is in a "Closed" State. • Enter "0" and then press "Send" in the Serial Monitor. • Immediately enter "1" and then press "Send" in the Serial Monitor. • Place an object between the "object in the door" sensors. • Immediately enter "1" and then press "Send" in the Serial Monitor. • Wait ten (10) seconds. • Press the "Open" sensor. • Enter "1" and then press "Send" in the Serial Monitor.
Expected Result:	Current State 3, then Current State 3, and then Current State 2.
Actual Result:	Current State 3, then Current State 3, and then Current State 2.
Pass/Fail:	Pass

Garage Door Closing with Interrupt	
Description:	This will test the garage door going from an "Open" state, to a "Closing" state, to an "Open" state, with an object placed in the door. First make sure the garage door is in an "Open" state.

	• Enter “0” and then press “Send” in the Serial Monitor. • Immediately enter “1” and then press “Send” in the Serial Monitor. • Place an object between the “object in the door” sensors. • Immediately enter “1” and then press “Send” in the Serial Monitor. • Wait ten (10) seconds. • Press the “Open” sensor. • Enter “1” and then press “Send” in the Serial Monitor.
Expected Result:	Current State 3, then Current State 1, and then Current State 0.
Actual Result:	Current State 3, then Current State 1, and then Current State 0.
Pass/Fail:	Pass

5 Overview

Provides a brief overview of the testing approach, testing frameworks, and general how testing is done to provide a measure of success for the system.

5.1.1 iPhone Application

Testing on the iPhone application was done by simulating communication with a base station and resolution server. The testers manually responded to requests from the application with JSON responses designed to get the desired result from the application.

5.1.2 Base Station

The Base Station is tested by sending it requests and looking at the onboard logs to determine how it handled the request. To facilitate message passing, a dummy client was written which can handle batch requests and record returned data.

5.1.3 Resolution Server

The resolution server was tested using a script to simulate the iPhone and Base Station. It made all of the requests using both valid and invalid data.

5.1.4 Devices

The device controllers can be tested using the standard Arduino testing tools and serial communication programs, like PuTTY.

5.2 Dependencies

5.2.1 iPhone Application

The iPhone application depends on Apple's Cocoa Touch framework.

5.2.2 Base Station

The base station depends on the same version of Python as the Resolution Server, TCP for internet communication and PySerial for USB communication. The web configuration frontend requires PHP 5.4 or later and an Apache server with PHP enabled.

5.2.3 Resolution Server

The Resolution Server depends on Python 2.7 and the SQLite database.

5.2.4 Devices

The devices will each require an Arduino control board or that the device itself implement the necessary logic and communication to a base station.

5.3 Test Setup and Execution

5.3.1 iPhone Application

The iPhone application was tested using programs that allowed the testers to receive requests from the application and respond with JSON data designed to trigger the desired result in the application.

5.3.2 Base Station

A small python script was created to send command strings to the base station. This script reads commands, one line at a time, from a plaintext file and prints the server's response. With redirection, it is possible to generate files for executing and validating batches of commands. This dummy client is available in the dummy directory of the base station, and is available for use as a sample for communicating in addition to its test role.

5.3.3 Resolution Server

The resolution server was tested using a Python script that simulated requests from the iPhone application and the base station.

5.3.4 Devices

The garage door controller was tested using the Arduino serial monitor.

6 Development Environment

6.1.1 iPhone Application

The iPhone application must be developed on Mac computer using Apple's Xcode IDE. Xcode can be downloaded from Apple's website for free after creating a free Apple developer account. The application can be developed and tested on the simulator with a free account, but to test the application on hardware and distribute it a paid subscription to the Apple iOS Developer Program is required.

6.1.2 Base Station

The Base Station was developed in a POSIX-compliant environment which had Python with PySerial available, using a plain text editor.

6.1.3 Resolution Server

The Resolution Server is being developed on a Mac using a plain text editor, but any with Python 2.7 installed will work as well.

6.1.4 Devices

The Arduino based device controllers were developed using the free Arduino development tools.

6.2 Development IDE and Tools

6.2.1 iPhone Application

The Xcode IDE and the iOS Simulator were used to develop the iPhone application.

6.2.2 Base Station

Development was done using a plain text editor.

6.2.3 Resolution Server

Development was done in a plain text editor.

6.2.4 Devices

The Garage Door controller was written in C++ using the Android IDE.

6.3 Source Control

Github was used for source control. It has a Documents directory to store documentation and a src directory for project code. There is also a package directory which contains the various packages which are deliverables for this project.

6.4 Dependencies

The system depends on the iPhone API and Python with the PySerial module. The optional web frontend depends on PHP 5.4 or later and apache.

6.5 Build Environment

6.5.1 iPhone Application

The iPhone application must be built using Xcode on a Mac.

6.5.2 Base Station

Python 2.7 with PySerial module is necessary for building. The web frontend is built on Apache and PHP 5.4 or later.

6.5.3 Resolution Server

Python 2.7 or a compatible interpreter and the SQLite database are necessary.

6.5.4 Devices

The garage door controller can be built on any system using Arduino software.

6.6 Development Machine Setup

6.6.1 iPhone Application

The iPhone application was developed on a Mac with Apple's developer tools installed.

6.6.2 Base Station

The Base Station was developed in various Linux distributions using vim, emacs and cat. The machine had Python 2.7, PySerial, Apache and PHP installed.

6.6.3 Resolution Server

A computer with Python 2.7, SQLite and a text editor are required to develop the resolution server.

6.6.4 Devices

The Garage Door controller was developed using the Arduino IDE.

7 Release | Setup | Deployment

7.1.1 iPhone Application

The application must be distributed through Apple’s AppStore to be installed on users’ phones.

7.1.2 Base Station

To setup a Base Station, the software must be installed on a computer and the network must be configured to allow incoming connections to reach the Base Station.

7.1.3 Resolution Server

The Resolution Server software should be installed on a computer with a domain name or a static IP address so the iPhone application and Base Stations know where to find it.

7.1.4 Devices

The garage door controller should be installed on the Arduino microcontroller, and the Arduino should be connected to the base station using an A/B USB cable.

7.2 Deployment Information and Dependencies

7.2.1 iPhone Application

The application can be deployed to any iOS device by downloading it from the AppStore. For development, it can be deployed to a registered development device using Apple’s developer tools.

7.2.2 Base Station

The Base Station can be deployed to any environment which can interpret Python 2.7 or later and which has the ability to host web sites in PHP.

7.2.3 Resolution Server

The resolution server can be deployed on any computer with Python 2.7 and SQLite installed. This computer must have fixed host name such as a domain name or a static IP address so the base station and application can consistently find it.

7.2.4 Devices

The garage door controller can only be installed on an Arduino microcontroller with both a prototype shield and a relay shield.

7.3 Setup Information

7.4 System Versioning Information

There is currently no formal versioning system, but since this release is largely complete, an arbitrary initial release value of 0.8 will be selected. Releases will be divided into major and minor releases, with even-numbered minor releases indicating a “stable” variant and odd-numbered indicating “unstable” or “beta” releases. Major releases on a maintained system will occur every two years or as specific milestones are met, while minor releases should contain bug fixes and similar modifications.

7.5 Getting the iPhone App

To get the Remote Home App on your iPhone, download it from the AppStore on your phone or download it from iTunes on your computer and sync your phone.

7.6 Setting up your Base Station

To setup your Base Station plug the power cord in and then plug the network cable into your router. You will have to configure your router’s firewall to allow your phone to connect to your Base Station when you are away from your network. If you do not know how to do this, consult your router’s user manual or you can contact us at (555) 123-4567 and one of our representatives will help you.

7.7 Adding a Base Station to Your Phone

The first time you open the Remote Home App on your phone you will be asked to add a Base Station. Fill in the text boxes with the Station’s serial number, which is printed on your Station, a memorable name for your device(it doesn’t matter what it is), and the password for your Station.

Appendix

I. List of Figures

II. Supporting Information and Details

A. Communication Protocols

1. Bidirectional iOS to Resolution Server Communication

This communication protocol defines the messages that will be passed between an iOS client and the Resolution Server. This will allow the iOS client to loop up IP address for a base station from a serial number.

a) *DDNSConnected*

This message is passed when the Resolution Server acknowledges a connection from a iOS client.

```
{ "DDNSConnected": [  { "Connected": true } ] }
```

b) *HRHomeStationsRequest*

This message is sent from the iOS client to the Resolution Server. This message is a request for IP addresses based on serial number. “(StationDID)" field will be replaced by a base station serial number.

```
{ "HRHomeStationsRequest" : [ { "StationDID" : "(StationDID)"}, { "StationDID" : "(StationDID)"}, ... ] }
```

c) *HRHomeStationReply*

This message is sent from the Resolution Server to the iOS client. This will tell the iOS client the association between serial numbers and IP addresses. “(StationDID)” is the station serial number. "xxx.xxx.xxx.xxx" is the IPv4 address of the base station. If the station cannot find the IPv4 address it will fill the "xxx.xxx.xxx.xxx" field with a null.

```
{ "HRHomeStationReply" : [ {"StationDID" : "(stationDID)", "StationIP" : "xxx.xxx.xxx.xxx"},
{"StationDID" : "(stationDID)", "StationIP" : null}, ... ] }
```

2. Unidirectional Base Station to Resolution Server Communication

This communication protocol defines the messages that will allow the Base Station to update Resolution Server with its current IP address.

a) *HRHomeStationUpdate*

This message is sent from a Base Station to the Resolution Server. It contains the station's unique identifier and its current IP address.

{"HRHomeStationUpdate":{"StationDID":"(StationDID)","StationIP":"(xxx.xxx.xxx.xxx)"}}

3. Bidirectional iOS to Base Station Communication

This communication protocol defines the messages that will allow the iPhone application to connect to a base station, get information about the devices connected to a base station, and send commands to those devices.

a) *HRLoginPassword*

This message is sent from the iOS application to a base station to make an initial connection to the base station.

{"HRLoginPassword" : "(password)"}

b) *HRLoginSuccess*

This message is sent from base station to the iOS application in response to an HRLoginPassword request. It tells if the login was successful, the number of devices, and list of devices including their type, status, and serial number.

{"HRLoginSuccess" : (true|false), "HRDeviceCount" : (int),"HRDeviceList":[{ "DeviceName":"(name)", "DeviceSerial":"(serial)","DeviceType": (int), "ErrorCode": (int)},…]}

c) *HRDeviceRequest*

This message is sent from the iOS application to the base station to send commands to devices and also from the base station to the iOS application to return data to the application. It is a general purpose message that will work with any device.

{"HRDeviceRequest": {"DeviceID":"(stringID"), "Password":"(string)", "Type":"(stringType)", "Data":"(stringData"),"Human Message":"(stringHumanReadableText)"}}

II. Progress | Sprint Reports

This section will contain a complete list of all of the period progress and/or sprint reports which are deliverables for the phases and versions of the system.

A. Sprint 1 Progress Report

Sprint Report 1
Team Name: Remote House
Team Members: James Wiegand, Christoper Jensen, Joshua Kinkade, Brian Vogel
Sponsor: L-3 Communications
Description: L-3 Communications is a company that develops, command and control, avionics, and communications technology for commercial and government customers.
Project Goal: The goal of this project is to allow people to control devices in their house using an application on their smart phone.

Needs:
• An application to send commands to devices remotely
• A way to physically control devices
• A way to connect the two things listed above
Project Overview: We are creating a system that allows users to control devices in their house, such as a garage door opener, from an application on their iPhone. Our system will have a gateway in the house that coordinates communication to and from the app with the numerous devices in the house. The devices will be connected to the gateway using Bluetooth wireless communication. We will use a centralized dynamic domain name server to allow users to connect their app with the specific gateway in their house. Our software is designed to allow new types of devices to be easily included in the system.

Project Environment:
Mobile Application: We are developing our mobile application for iOS, primarily targeting iPhones. We may also create and Android application if we have time.
Gateway: Our gateway will be a small Intel based Linux server.
Device Controllers: The controller for each device will be an Arduino board with a Bluetooth shield to communicate with the gateway.
DNS server: The DNS server will store the IP addresses of every gateway, so the app can communicate directly with the gateway.

Project Deliverables:
• iPhone application
• Gateway software
• DNS software
• Prototype Arduino control board
• Documentation

Backlog:
• Server Configuration
• iOS Register Device
• Communications Protocol

• iOS Main View
• iOS Status View
• iOS Garage Door opener Controller View
• Garage Door opener

B. Sprint 2 Progress Report

Sprint Report 2
Christopher Jensen James Wiegand Joshua Kinkade
Brian Vogel
November 9, 2012

1. Hardware
All hardware components except for the Bluetooth Shield have been acquired for the garage door. An additional Arduino will be necessary for the sprinkler system, but one of the requisite valves has been purchased. Wiring will commence and a working opener over Serial communication should be ready by theend of next sprint.
2. iOS Interface
A rudimentary prototype for the opener app has been completed. The app will be updated when more information on server-phone communications is available for the iOS developer to work with.
3. Gateway/Server
No progress has been made on the Gateway or Server interface for this sprint, in favor of getting the iOS and hardware applications resolved. Gateway will not likely see further development until after initial hardware and full communication protocol is complete.

C. Sprint 3 Progress Report

Sprint Report 3
12/6/2012
James Wiegand
Joshua Kinkade
Christopher Jensen
Brian Vogel

iOS
James continued working on the top level framework for the iOS application and created a prototype for the first run of the device where the user will register their device with a base station.

Resolution Server
Josh began working on the resolution server and created the first draft of the base station to resolution server communications protocol that allows base stations to update their IP addresses.

Base Station
We did not made any progress on the Base Station this sprint.

Hardware

Chris continued experimenting with a garage door opener to find a way to control it with an Arduino board.

D. Sprint 4 Progress Report

Sprint Report 4
Prepared by:
Joshua Kinkade
Christopher Jensen
Brian Vogel
James Wiegand
South Dakota School of Mines and Technology
Spring 2013
February 8, 2013

Overview
This sprint report is to give an update on the progress of the Remote Home team on the development of home devices controlled by a user's smart phone. This report will cover a recap of progress that has been made last year. Additionally, it will cover the progress made this year in the top level iOS framework, the iOS mobile application, the domain name resolution server, and the garage door hardware. Finally this report will give a list of prototypes for this sprint.

Sprint 1-3 Progress
Last year the Remote Home team was given the task of developing a system that allows users to control devices in their home, such as a garage door, from an application on their iPhone. The system was designed to have a "base station" in the house that coordinates communication to and from the application with numerous devices in the house. The system was also designed to have a centralized dynamic domain name resolution server to allow users to connect their application with a specific base station in their house, without the user needing to have a static IP address. The devices in the house will be connected to the base station using Bluetooth wireless technology. The system was designed to allow new types of devices to easily be included into the system. James started developing a top level iOS framework that will link a user with a specific base station and all of the devices associated with the base station. The framework also manages the devices associated with the base station and allows the user to register new devices and remove devices that are currently on the system. Joshua created a prototype garage door opener application that the user will use when controlling their garage door. He also began developing a dynamic domain name resolution server that will store the IP addresses of every base station, so that a user's application can communicate directly with the base station located in their home. He finally started developing a communication protocol that will allow a base station to update its IP address on the dynamic domain name resolution server. Chris started by gathering the necessary hardware for the project, e.g. garage door opener, and Arduino microcontroller. After gathering the hardware Chris then began working on controlling the garage door opener over serial communications.

iOS Framework
James completed developing and debugging the top level frame work over the Christmas break. Base stations and devices can now be registered on the system.

Resolution Server

Joshua continued developing the dynamic domain name resolution server, running into some issues when connecting to the iOS framework. These issues are currently being resolved and the resolution server should be completed before the next sprint.

Hardware
This sprint Chris acquired the Bluetooth shield, the final piece of hardware need required for the project. Chris also successfully controlled the garage door opener over serial communication using the Arduino. This shows a proof of concept in opening a garage door using an Arduino. The next step is to control the garage door opener using the Bluetooth shield.

Prototypes
The Remote Home team produced two prototypes this sprint. The first prototype was the iOS framework developed by James. The client is now able to register both base stations and devices on the system. The second prototype is a garage door opener controlled over serial communication using an Arduino.

E. Sprint 5 Progress Report

Sprint Report 5
3/15/2013
James Wiegand
Joshua Kinkade
Christopher Jensen
Brian Vogel

iOS
Top-level framework and Garage Door Controller are in beta state; final testing to commence in coming sprint.

Resolution Server
Resolution Server is in alpha state; initial testing is underway, and some revisions are expected as integration begins.

Base Station
Base Station is in alpha state, with a threaded server and some basic configuration files used to control users and configure device interfaces. Currently TCP/IP and HCI interfaces are not supported.

Hardware
The garage door opener has entered a beta state, with the ability to open a garage door over USB. It was discovered that the HCI device does not perfectly integrate with the relay board, so some minor modifications will be necessary to use Bluetooth to communicate with the device.

F. Sprint 6 Progress Report

Sprint 4 Report
April 12, 2013
Chris Jensen
Joshua Kinkade

Brian Vogel
James Wiegand

iOS
James did unit testing for the iOS application framework. Josh modified the garage door view controller to work with the revised garage door messages and did unit testing for the garage door view controller.

Base Station
Chris completed the base station, including the web based front end, and did unit testing for the base station.

Garage Door Controller
Brian converted the garage door controller to a state based system and completed the controller. He also did unit testing for the controller.

Resolution Server
Josh has added error handling to the resolution server and has completed unit testing ofthe resolution server.

Integration
The team has integrated the different components of the system and has done most integration testing. Some additional testing will be done this weekend.

Poster
The team has put the poster for the design fair together.