Navigation and Automation with the Pebble Smartwatch

Nico Ekasumara, Parker Finch, Isaiah Leonard
Williams College

1 Introduction

In this paper, we describe our final project for the Distributed Systems course: a system that uses a simple protocol to connect a Pebble Smartwatch, Android device, and (at least one) Arduino-controlled device to provide remote device automation, as well as a location-based automation and navigation system. Our primary goal in designing this system was creating protocols and user interfaces that make device automation relatively simple; this system serves as a demonstration of the feasibility of connecting and automating arbitrary devices (not only those which are built to be web-connected) via a simple interface. In addition, our system is designed to allow for location-based (geofenced) device automation: users can associate automated devices with locations, and then when they reach (or near) that location, they can use the Pebble watch to set the state of the associated device. Finally, as another example of how the Pebble watch (or similar devices) can make interaction more convenient for the user, we provide navigation: the user can select a destination (either a previously saved location or one selected from a map) on his Android device, and the Pebble watch will then guide him to his destination.

The remainder of this paper is organized as follows. Section 2 provides an architectural overview of our system, describing the infrastructure of our code and how the system components communicate. Section 3 evaluates our system, focusing on performance, usability, and extensibility. In Section 4 we discuss some possible extensions that would provide increased functionality. Finally, Section 5 summarizes our system and experiences.

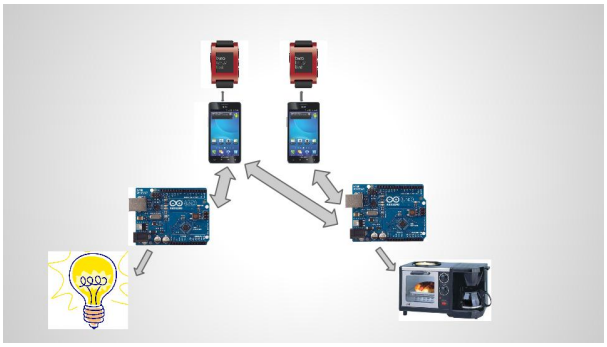


Figure 1: A diagram showing how the hardware components interact

2 Architectural Overview

2.1 Hardware Components

Our system consists of four main types of hardware components: a smartwatch, smartphone, Arduino board, and automated device. Figure 1 gives a visual representation of how these components interact.

2.1.1 Pebble Smartwatch

The first hardware component is the Pebble watch itself. Pebble smartwatches have relatively limited capabilities on their own, and rely heavily on a paired smartphone (either Android or iPhone). Indeed, the watches have a few basic sensors, such as an accelerometer and a frustratingly-disabled mag-

netometer (which could be used to get compass readings), but have no data I/O beyond the Bluetooth connection to a smartphone. Therefore, the main use of the Pebble, at least for our project, is simply to provide a convenient interface for the user, allowing the user to follow directions to a chosen location or automate a device when he/she reaches one of the geofenced locations without having to take out a smartphone. The watch simply communicates with the phone (in our case an Android smartphone) which performs all calculations and networking.

2.1.2 Android Device

Our second main component is an Android smartphone running our Android application (we used a Samsung Galaxy S4 for our testing, but any phone with Android version at least 4.0 would work with our application). The Android device performs the vast majority of the calculations and communications in our system: calculating and providing directions, communicating with the Pebble watch to display relevant information and receive user input, and communicating with the Arduino servers to perform automated tasks. Given the substantial amount of calculation and the wide range of jobs that the smartphone has to perform in our system (as well as our desire to present a nice UI on the phone as well), we chose to implement an actual Android application, rather than simply use a Javascript program that would provide minimal smartphone functionality and interaction for the user. Though an iPhone could work equally well in this capacity, for this project we chose to use Android because we had more experience with the platform.

Communication between the Pebble watch app and the Android phone is handled by sending dictionaries. In the Android application, we store the values we want to send in the dictionary; on the watch, we then get the values out of the sent dictionaries by using the key. One consequence of this protocol is that the developer for the phone and watch app must be coordinating with each other (or the same person), since one must know the used keys in order to obtain the desired values.

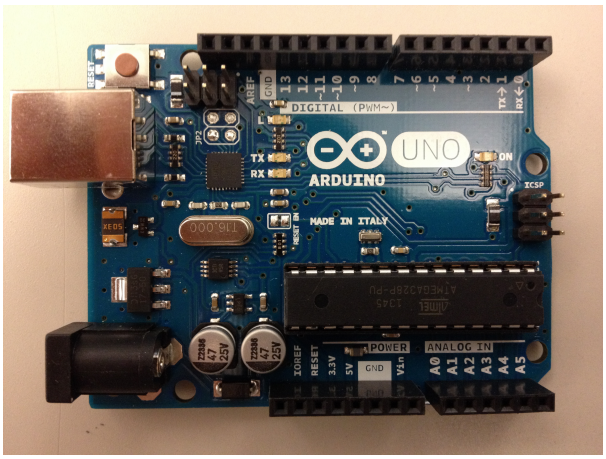


Figure 2: Arduino Uno board.

2.1.3 Arduino Uno with Ethernet Shield

Arduino is an open-source, single-board microcontroller that allows for relatively easy interaction with other electronic objects. We used an Arduino Uno (Figure 2) for our project, but other Arduino microcontrollers could also be used. In addition, to provide internet connectivity and allow for remote communication and activation, we used an Arduino Ethernet Shield, which sits on top of the Arduino Uno microcontroller and gives the Arduino the ability to act as a web server. The Arduino receives commands from the Android phone and then controls the automation of an attached device, such as a light switch, fan, or wonderful, high-quality breakfast machine.

2.2 Android Application

The main software component of the system is our Android application: `Distributed Directions`. It tracks the user's location, receives user input, and communicates with both the Pebble smartwatch and Arduino servers. `Distributed Directions` has five main components, each with its own corresponding button on the application's launch screen, as shown in Figure 3.

2.2.1 Navigation

When a user selects our navigation component, denoted by the `Get Directions` button on the ap-

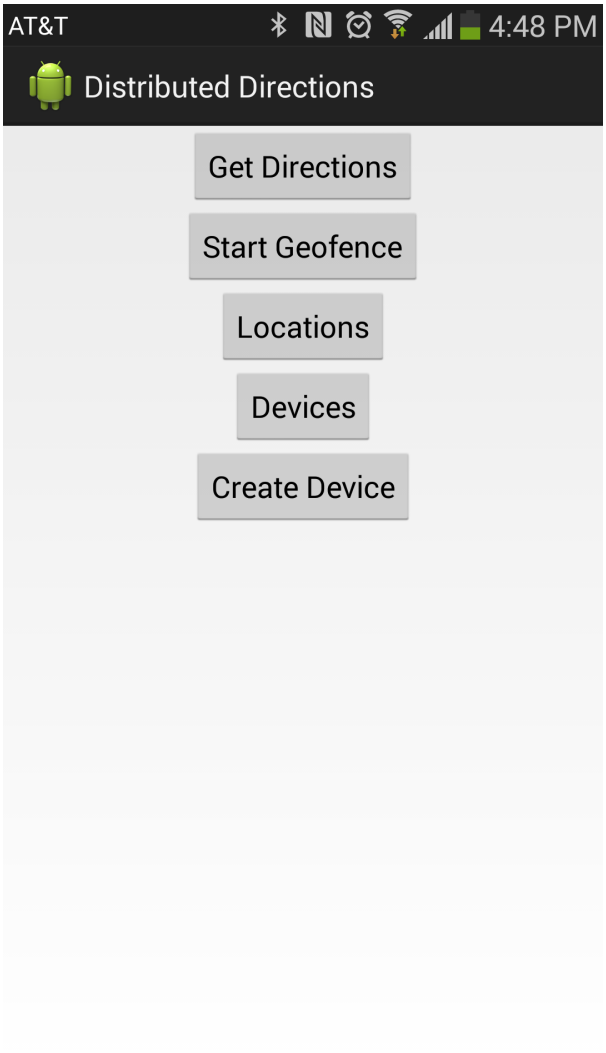


Figure 3: `Distributed Directions` start screen.

plication start screen, we start the `MapsActivity` in the Android application and then send a signal to the watch app to turn the watch app on. The user is shown a map in the phone application and is prompted to select a destination by long-pressing on the map. The phone stores that location (recording the latitude and longitude) and starts the location manager, which periodically updates the phone's current location. After computing the distance and direction to the destination from the phone's current location, the application sends those two values back to the watch. For this navigation app, the phone app sends a dictionary with two items, the first is the distance and the second is the angle to the destination. The watch then shows the distance and draws an arrow pointing in the appropriate direction.

One problem we had with the navigation was that the current Pebble SDK doesn't support the watch's magnetometer yet, so we can't use the watch to determine what direction the user is facing. We tried to use previous locations in order to find the direction of recent movement, but the results were very inconsistent because the phone's GPS is not all that accurate and slight changes to your movement yield inaccurate directions. To resolve this issue, we decided to just give directions with respect to north. Hence, our watch app layout gave a picture of a compass with a line pointing in the direction of the destination with respect to the north. This watch app layout is shown in Figure 4.

2.2.2 `AutomatedDevices`

`AutomatedDevice` objects are used to store information that allows the Android application to communicate with an Arduino server connected to a device of some sort. Therefore, the `AutomatedDevice` must record an IP address and port number of the corresponding Arduino server, as well as the list of possible states for the connected device.

To create a new `AutomatedDevice` in the application, the user must press the `Create Device` button on the start screen. This launches a page shown in Figure 5 in which the user puts in the name, IP address, and port number of the `AutomatedDevice`. When the user presses the `Create Device` button, we create a `DeviceRequest` to send a message ('setup') to the given address. Assuming that there is in fact an Arduino server running our protocol at that address, it will send back a message which gives a list of possible states for that device; for example, our oven gives back off and on ('Off;On'), but this infrastructure can handle devices with any number of possible states (eg: a fan with multiple speeds). These `PossibleStates` are stored in the `AutomatedDevice`.

Connection to the Arduino uses a `DeviceRequest` object, which extends Android's `AsyncTask` because network tasks are

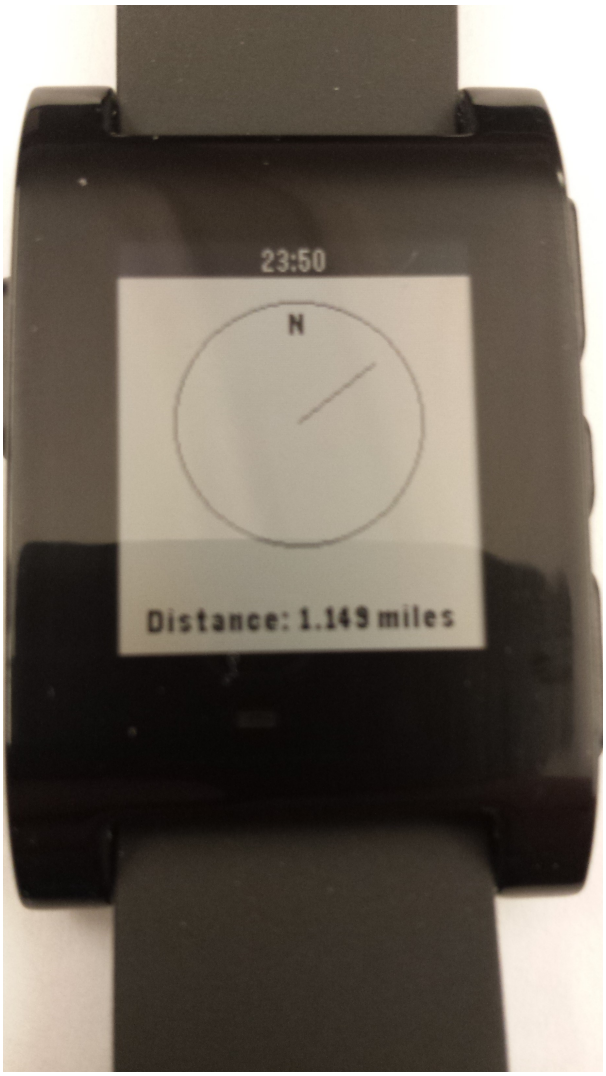


Figure 4: Directions on the Pebble watch

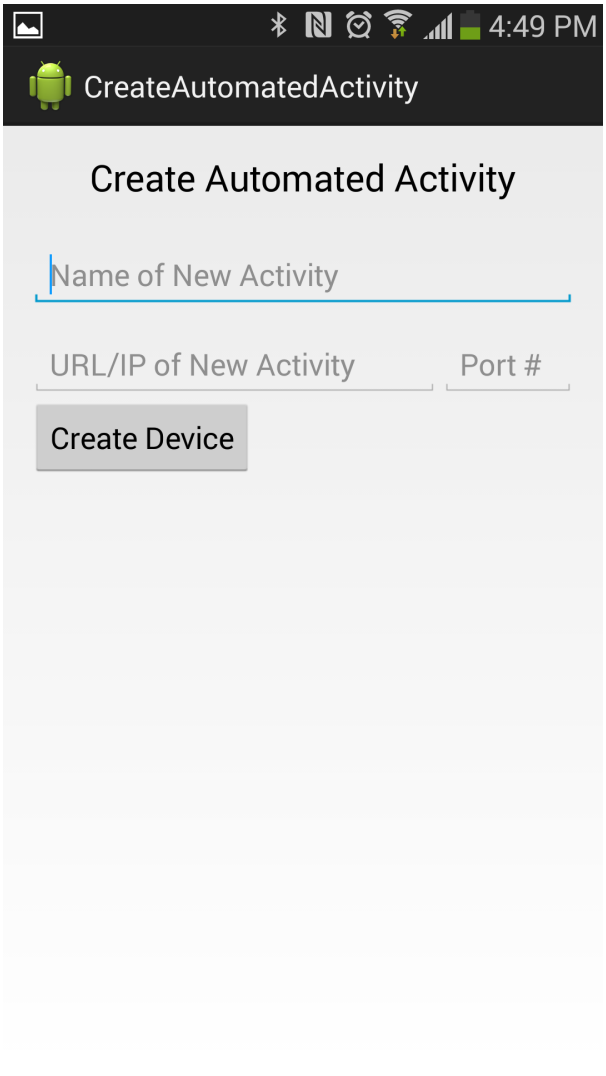


Figure 5: Creating a new device

supposed to be performed in the background. `DeviceRequests` simply take an IP address, port number, and message; they send the message to that address and return the server's response for processing by the application.

To make automation easier, we also implemented what is effectively a remote-control mode that allows the user to set any saved `AutomatedDevice` to any of its `PossibleStates`. The user accesses this mode by selecting the `Devices` button on the start screen, giving a list of all the saved `AutomatedDevices`. Selecting one of the devices brings up a UI for that device specifically, allowing her to select a `PossibleState` and set the device to that state.

2.2.3 `SavedLocations`

The Geofencing portion of our application requires a set of `SavedLocations` with associated `AutomatedDevices`. To add a new location, the user can open up the map by pressing the `Start Geofence` button, and long press on a location in the displayed map. Clicking on a location brings the user to a list of locations, where the new location is added with a name "New Location". The user can then click on that New Location in the list, redirecting him to an activity which allows him to change the name of the location and its as-

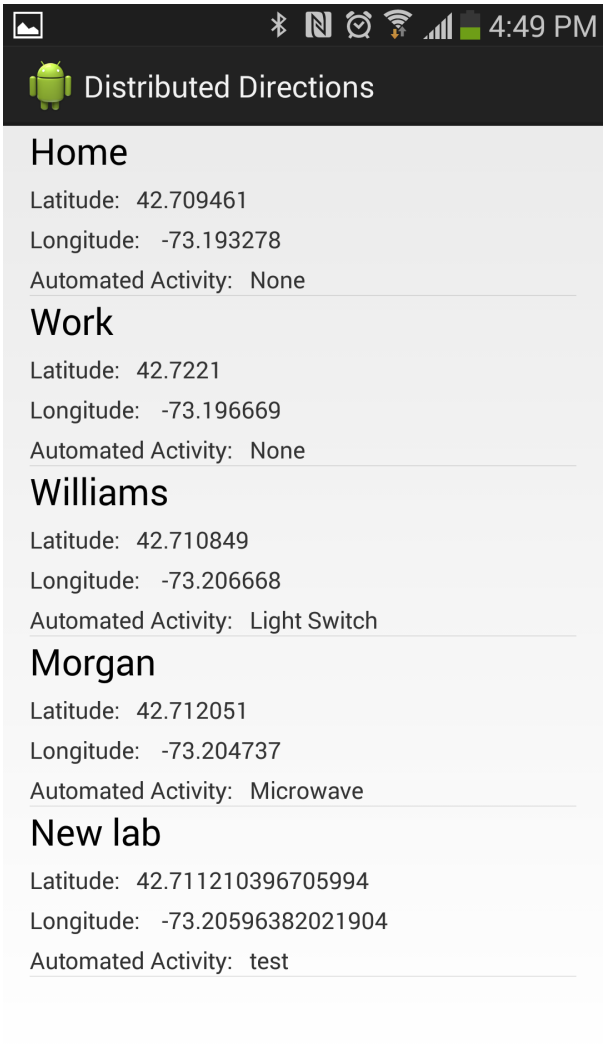


Figure 6: List of `SavedLocations`

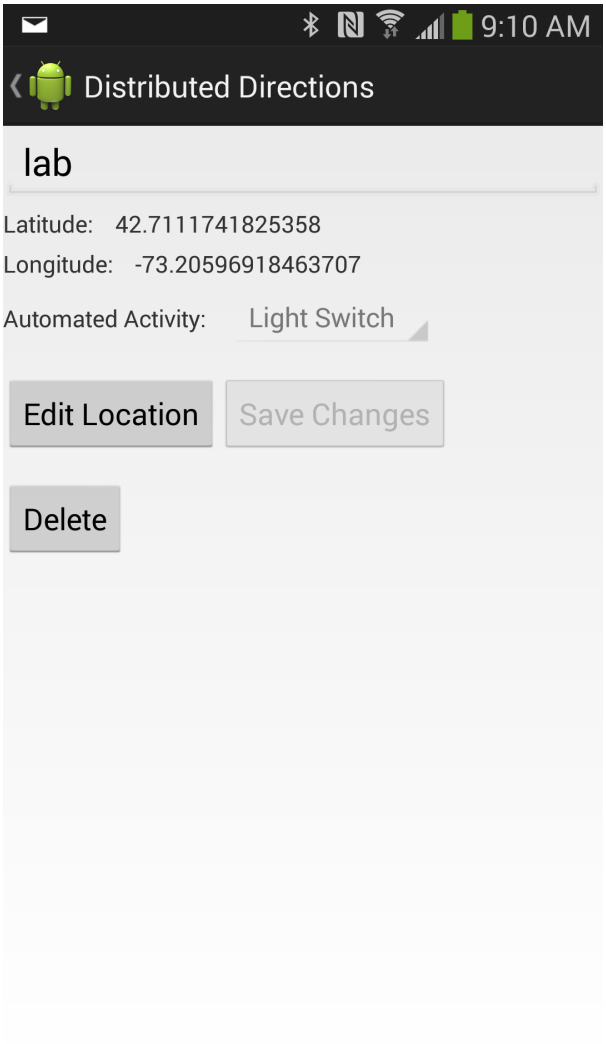


Figure 7: Editing a `SavedLocation`

sociated `AutomatedDevice`. Figures 6 and 7 show the list of of locations and how the edit location pages look. We stored each of these locations in a `SavedLocation` object, which stores the name, the device name, and the location itself (recorded as a latitude and longitude). We can use this `SavedLocation` object to easily refer to a location and find the device it is associated with.

The user can also edit or remove `SavedLocations` by clicking on the `Locations` button on the start screen. This brings her to the list of `SavedLocations`, allowing her to select one for editing or deletion.

2.2.4 Geofencing Service

The geofencing portion of our application makes use of all three hardware components in our system: the watch, the phone, and the Arduino server. The geofencing service begins when the user clicks the `Start Geofence` button on the start screen. As with the navigation service, the application starts its location manager which keeps track of the phone's current location.

Whenever the current location is updated, we iterate through the list of `SavedLocations` and calculate the distance between each of those locations and the current location. If any of the distances is small enough - our geofence is a circle with radius

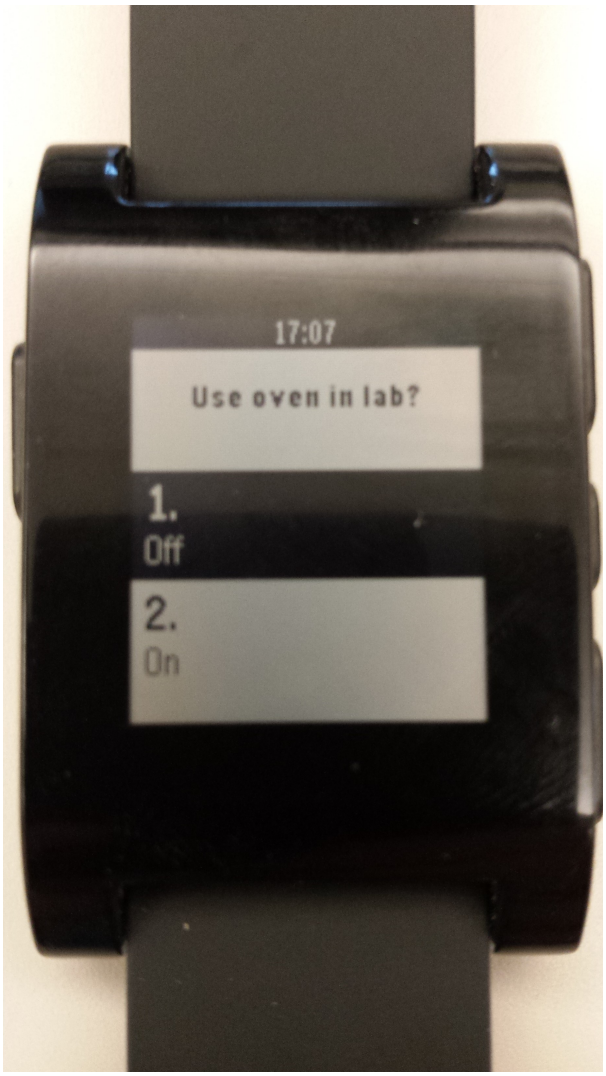


Figure 8: The user has crossed a geofence near one of his saved locations, and now can choose whether to start the oven in lab.

.05 miles, but this radius is arbitrary and we can change its size quite easily - the application sends a message to the watch app asking the user what command (if any) to send to the `AutomatedDevice` associated with that location (and connected via an Arduino server).

To display these options to the user on the Pebble watch, we send a dictionary to the watch app with the name of the location, the name of the `AutomatedDevice`, how many possible states the device has, and all of those possible states for the device. Then, in the watch app, we display those possible states as a list of options for the user. The watch app is shown in Figure 8.

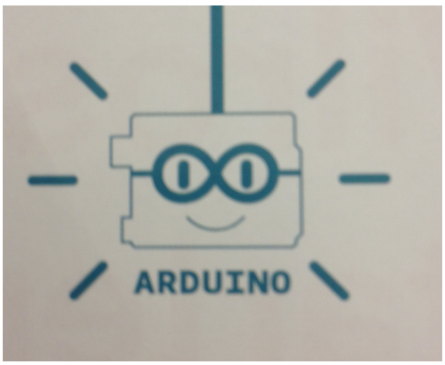


Figure 9: A happy Arduino.

When the user selects an option by pressing the center button on the Pebble, the watch application creates a dictionary with the name of the location and the chosen option. We send this dictionary to the phone application which can now find the appropriate device and the user's selected state for that device. Since we store the IP address and port number in the device object, we can then use a `DeviceRequest` to connect to the appropriate Arduino server and send the chosen option. The Arduino server will handle this request, set its connected device appropriately, and then respond with an 'OK' message.

2.3 Arduino Setup

In our system, we used an Arduino board to realize action in the physical world. While we used it for a very specific purpose, namely operating a 3-in-1 breakfast maker, it is important to stress the flexibility of our system. Throughout the description of our

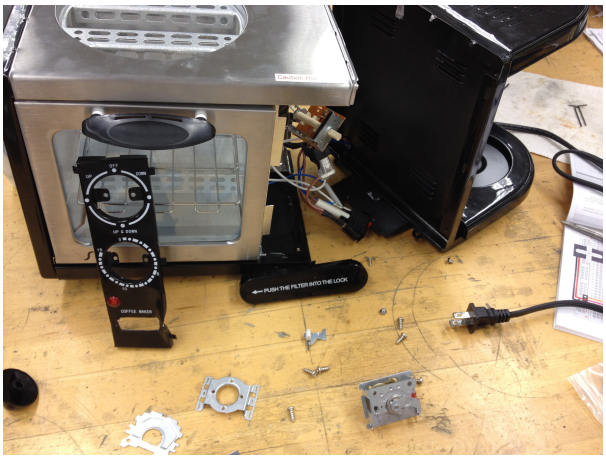


Figure 10: A distributed breakfast machine.

setup, we will be sure to note which aspects of our design are widely applicable.

We used an Arduino Uno board with 32k of memory and 16MHz processor, pictured in Figure 2. In order to connect the Arduino to the Internet we used an Ethernet Shield, which interfaces directly with the arduino. This requires that our arduino is connected with an Ethernet cable, but is much less expensive than a wifi shield.

To cook our breakfast we used a Sunpentown BM-1107 Stainless-Steel 3-in-1 Breakfast Maker, which we modified slightly to suit our needs. We removed the human-readable control panel to expose the wiring. See Figure 10. We disconnected the wiring from the manual timer and attached it to a switch controlled by the Arduino board. This allows our Arduino to turn the toaster/frying pan on and off.

2.4 Arduino Server

We implemented a simple server for the Arduino. This involved following the protocol we designed for the Android application.

2.4.1 Communication Protocol

The Android phone initiates communication, and requires the IP address of the Arduino. It first sends a message consisting entirely of the string ”setup” follows by a newline character. In response the Arduino sends ”OK:”, follows by a semicolon-delimited list of its possible states. For example, an Arduino controlling an adjustable light might be able to be in one of four possible states: `Off`, `Low`, `Medium`, or `High`. In this case the Arduino’s response would be the string ”OK:Off;Low;Medium;High”.

The Android application uses the Arduino’s response to build a list of possible actions, which can then be displayed to the user. When the user selects an option, say ”Medium”, the string corresponding to the chosen action, in this case ”Medium”, (with a newline character) is sent from the phone to the Arduino server. The Arduino then performs the requested action.

2.4.2 Server Implementation

Our server itself is not complicated. It waits for an incoming connection, and then reads characters until a newline is sent. Then it compares the received characters with a list of predefined options and performs the corresponding action. If the received message does not match any of these predefined actions, then the server simply closes the connection. We have found that this simple server model is easy to implement, reliable, and sufficient for our purposes. By providing a list of arbitrary actions, we maintain enough flexibility to automate a wide variety of appliances.

3 Evaluation

Given that our primary goal for this project was to create a simple, usable, easily extensible protocol for device automation (specifically for devices not designed to be automated) and then to provide prototypes for that protocol, the evaluation will focus on a few main factors, primarily functionality and reliability (ie: does it work), usability (ease of use), and scalability (how easy would it be to add new devices and protocol).

3.1 Functionality and Reliability

In our testing, our system’s functional performance was extremely good. We tested our system’s connection functionality by setting up an Arduino server with three led bulbs coordinated to

model a multi-state device with off, low, medium, and high settings (the code for this server is in `multi_state_server.ino`. We then tested this setup using both lab machines (connecting via telnet) and through a Pebble smartwatch and Galaxy S4 Android phone running `Distributed Directions`. In all cases, the system performed flawlessly: the application was able to add the server as a new `AutomatedDevice` and populate the device field appropriately, the watch provided a simple and functional interface for choosing between the possible states, and the Arduino server correctly processed each and every incoming message by setting the lights appropriately. However, there is a slight delay between the watch and the arduino of 1-2 seconds, as we need to send a signal from the watch to the phone then to the arduino.

To test the granularity of our phone's GPS location data (not really a part of our code, but still an important part of our system), we created `SavedLocations` at relatively nearby locations: the Paresky Center, Morgan, and Schow Library. These locations are separated by only about .07 miles, barely more than our geofence radius of .05 miles, but in our testing, our application was consistently able to differentiate between these locations appropriately. In walking back and forth between these locations several times, the correct notification was always sent to the Pebble watch, allowing control of the associated device.

The Arduino server was perfectly reliable for the duration of our testing. This is likely due to the fact that we have only been using a local network, and thus mitigate the possiblity of packet loss, and the simplicity of our implementation.

Our most important hardware has also proven to be reliable and fast: the breakfast machine heats up very quickly once power is applied, and made a fine egg and some toast in about five minutes.

3.2 Usability

While usability is obviously a subjective evaluation, we are generally very pleased with our system's usability. The `Distributed Directions` application is quite simple, allowing for easy creation of `AutomatedDevices` and `SavedLocations`. In addition, the remote-control mode accomplishes its purpose: it provides a simple interface to allow the control of any connected device. Finally, usability is where the Pebble smartwatch really earns its keep. Though it cannot do much calculation or really provide much data, it makes interaction much more convenient for the user: rather than having to take out her phone when reaching a `SavedLocation` in geofence mode, she can simply respond to the notification and select a device state from her watch. Likewise, being able to receive and follow directions without having to hold her phone out all the time is much more convenient.

3.3 Scalability and Extensibility

Overall, we think that the protocol we have developed accomplishes its goal of being scalable and extensible. To start, our communication protocol is quite simple, making the addition of new devices very easy: the corresponding Arduino server simply has to have a version of our code modified to give its possible states and then respond to those states by controlling the attached device appropriately. Moreover, our Android application supports easy addition of new devices, meaning that a large array of devices could be stored in the application, allowing for automation of nearly everything in your home. On that note, given how our application is designed, it would be relatively simple to add new communication protocols, making it possible to control devices which were already designed for web connectivity (this will be discussed more thoroughly in Section 4).

There is one major caveat that perhaps limits the overall extensibility of our system: the difficulty of setting up the hardware for an Arduino to control an appliance which is not designed for web-connectivity. For example, it's hard to envision many users being willing to get out the old hacksaw and have at their toaster ovens, removing the control panel, and then wiring up an Arduino with a relay to provide automated control. Though we certainly had a lot of fun doing so, it isn't necessarily the most practical solution. Nevertheless, we definitely do not view this as a death-knell for our system. First of all, the type of person who wants to automate their appliances is much more likely to be willing to hack their appliances to do so. Sec-

ond, as previously stated, our application could be easily extended to use other communication protocols for web-enabled devices; as more and more devices are designed with internet control in mind, less hacking would be required to control a wide array of appliances (even using multiple different protocols). Finally, different (better) Arduino hardware could greatly simplify the automation of existing devices by allowing for interaction with the normal (human-oriented) controls, rather than dealing with the wiring itself. For example, with higher power servos, we could have simply attached servos directly to the knobs on the control panel of our breakfast maker.

4 Future Work

There are many avenues of future work to explore. One is to automate a wider variety of appliances. We have designed a flexible protocol that allows an Arduino to control many types of appliances, but it is possible that varied hardware will make the realization of this automation significantly more difficult than we expect. In order to gain a better sense of the extensibility of our approach, we would like to implement our system with many more appliances.

We would also like to improve the security on our system. As of now, anyone with knowledge of the Arduino's IP address is able to control the corresponding appliance. This is a gaping security hole, for example, an Amherst student with knowledge of our breakfast machine could easily burn down the Williams Unix Lab. We would like to increase security while minimizing the added burden on the user. The large appeal of our system is its ease-of-use; if a user had to wait multiple seconds for key generation or enter a password for each request then the smartwatch would become merely a nuisance. One option is to have the user generate a password, which would be stored in the phone and hashed on the Arduino. The phone could then prepend every message with the password, and the Arduino would only obey messages with the proper password attached. There are obvious flaws with this approach. For one, a packet sniffer would immediately have access to the Arduino. Still, this would be easy to implement and a good first step to tighter security.

Another possible extension would be to implement a central server, likely on a personal computer (though it could also be on something simpler, such as a Raspberry Pi) to control all of the automated devices. First of all, this would make security much easier: there are countless libraries for data encryption and user authentication which are widely available for use. This would also further simplify use of our Android application for our users. Instead of having to input each `AutomatedDevice` separately, he could simply enter the IP address of the central server. The server could then respond with a list of its connected devices and their associated possible states.

Another task would be to extend our Android application with the ability to follow other protocols, such as INSTEON. This would allow us to control many automated devices in addition to those constructed with our implementation as described above. This would allow the Android phone to be a "universal remote" of sorts, controlling many varied appliances using different protocols.

One simple extension, which we mentioned above, would be to integrate the magnetometer into the directions application. The requirement of facing the watch North places a heavy onus on the user, with a magnetometer we would be able to give directions relative to the user's current orientation.

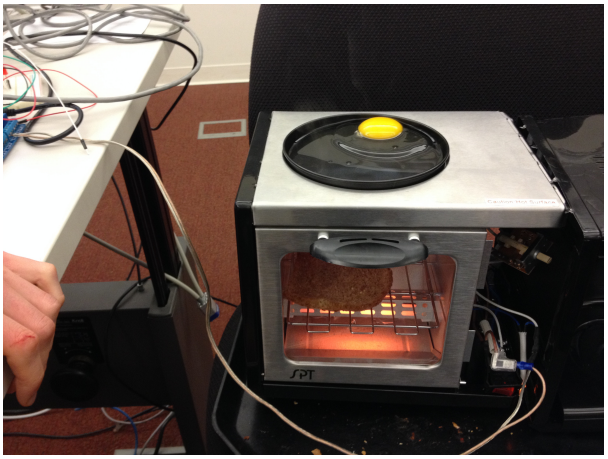
5 Conclusion

We have designed both a paradigm and a physical system for easy navigation and automation of appliances. By using the Pebble smartwatch we are able to relieve the user of the burden of interacting with their phone. This allows tourists to be less noticeable when following directions, while the geofencing feature reduces the overhead of activating automated appliances. We have found that our system is effective: we were able to reliable activate and deactivate a breakfast machine using only a smartwatch, and send a message to the smartwatch depending on your location. Moreover, the user-friendly interface allows anyone with an Arduino connected to their appliances to automate their home.

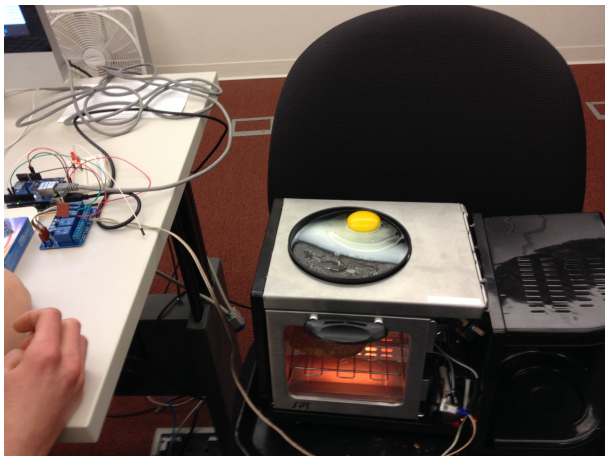
Almost as exciting as our experience and results is the potential for future work. There is still much

to be done in this field of minimalized automation. Our application and protocol leave a lot of flexibility, and inherent in this flexibility is a wide array of possibilities.

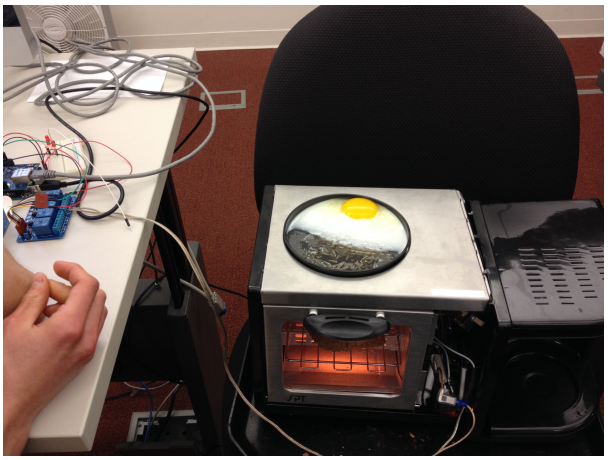
(a) Set your breakfast up before bed.



(b) Easily begin cooking your breakfast in the morning.



(c) Your breakfast will cook automatically!



(d) Breakfast finishes at the same time as your shower.

(e) Dig in and enjoy!

Figure 11: A time lapse of cooking breakfast.