Real-time Activity Observer (RAO)

Roman Kazarin, Aaron Magat, Oliver Townsend

1. Introduction

Every day, humans experience many different interactions and physical activities. At any point in time, each person can be categorized into a "state", based specifically on what action he/she is performing. Therefore, we can say that life can be modeled as a basic state machine (Figure 1.1), with each state being a human action like walking or biking and the connection between two states representing the transition. In this paper, we turn the spotlight to these state transitions specifically, as they can be harnessed to provide both useful information and/or actions to a human user. With the processing, networking, and sensing capabilities of modern smartphones, we can constantly "listen" to detect such transitions and in turn perform useful actions. We propose Real-time Activity Observer, or RAO, an application that monitors human state transitions using both low and high-powered sensors and lets users define the actions that should be performed whenever a specific state transition occurs. Our goal was not only to build an accurate, user-friendly application, but to also profile the application's energy usage. We implement RAO for the Android mobile operating system, testing the application on the Nexus 6P and LG Eclipse 2. We then evaluate the accuracy of our application. Finally, we profile the energy usage of the application using the Qualcomm Trepn Power Profiler [1], and observe the difference in energy usage between low and high-powered sensors, between different transitions, and between both test devices.

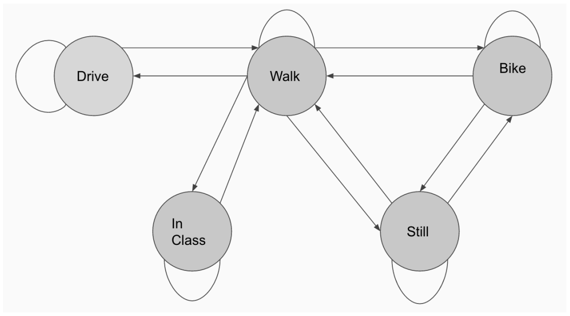

Figure 1.1.

2. Background

We are heavily influenced by the application If-This-Then-That, or IFTTT. IFTTT is a mobile application that allows users to customize and add on to their phone's functionality by creating "recipes" for certain software-based application events. For example, a recipe could be of the form, "If I post a picture to my Instagram, then save a copy of the edited version to my Dropbox." IFTTT however, does not have the option of creating "recipes" based on what a person is currently doing, or what he/she has transitioned to/from doing. IFTTT is mostly based on mobile application events, and does not utilize a device's sensors.

Our first goal was to take the user-friendly and intuitive concept of "recipe" from IFTTT and combine it with the use of sensors to detect state transitions. For example, instead of purely software based events, a recipe could be, "If I transition from walking to sitting and am in class, silence my phone". Detecting and acting upon such a recipe requires several steps. First, RAO needs to be able to identify the user's current state or activity. We utilize the Google Activity API for activity detection, which notably only uses a device's low-power sensors like the accelerometer and gyroscope. The application then needs to continuously check the user's current state, as well as remember what his/her previous state was. RAO also has to be able to detect whether a user is in class. We implement a geofence functionality, which lets the user tell us exactly where their class is. Therefore, if at any time the previous state was "walking" and the current state is "sitting", and the user's current location is within the aforementioned class, the application will proceed to silence the user's phone. While our application always uses low-powered sensors for basic activity detection, this recipe demonstrates the use of a high powered sensor (GPS) to provide the user with custom recipe options like setting up a geofence. We utilize high-powered sensors not only to add geofencing to a recipe, but to improve the accuracy of the detection performed by the low-powered sensors (see Section 4.2).

Our second goal was to study the energy consumption of RAO. While we constantly utilize the low-powered sensors (we receive an update from Google Activity API about every 5 seconds), using the high-powered sensors like GPS happens less frequently. In Section 4.1, we compare just how much more energy is used when performing a task that requires a high-powered sensor as compared to a low-powered one.

3. Solution

RAO was designed to initially have three main components: a component to handle user interaction, a component to handle activity detection, and a component to handle all other forms of sensing required by RAO. The user interaction component would allow users to create recipes (see Section 3.1) and display notifications created by the activity detection component. The activity detection component was responsible for monitoring user state and parsing the user created recipes in order to determine if any of them were triggered, and then taking the appropriate action if one was. The general detection component would handle any other necessary detection and would relay any results to the activity detection component.

We implement RAO in Java for use as an Android Application, and make extensive use of Google APIs throughout. While implementing activity detection, it was decided that a fourth component was necessary, one for location detection that was big enough to be separate from general detection.

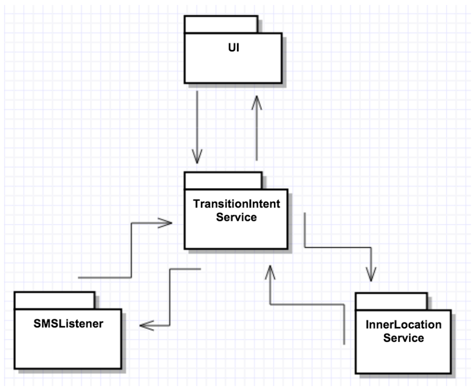

Figure 3.1: Architecture for RAO Android Application

3.1 Recipe

A recipe is the main form of user input for our application. A recipe allows users to dictate what sequence of physical actions will lead to a specified result, and thus a recipe can be seen as a combination of conditional statements and corresponding actions. Recipes take the form of an **IF** clause, which represents the user's current state, a **THEN** clause, which represents a user's future state, and a **DO** statement, which represents the action(s) to be taken. An example of a recipe can be seen in Figure 3.2, where the user has created a recipe that will drop a pin on a map when the user transitions from the current state, biking, to a new state, walking. In its ideal form, both the **IF** and **THEN** clauses will truly be comprised of conditional clauses which chain boolean expressions together, i.e. a user can choose their **IF** clause to be "IF: (Walking and Receive Text) or (Running and Receive Text)". Due to time constraints, recipes in RAO do not allow for this chaining, and thus **IF** and **THEN** clauses can be comprised of only one boolean expression.

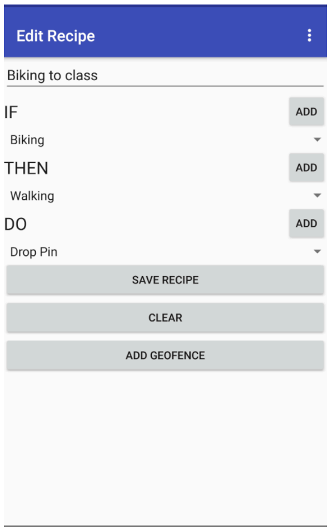

Figure 3.2: Example recipe being edited

3.2 User Interaction

The user interaction component is responsible for giving the user a way of creating and editing recipes. In addition, the user interaction component will display notifications to the user, signaling to the user that one of their recipes has been triggered.

This component is also the only one that will be visible to the user, and from the Android perspective will provide all the user interface. The RAO application exposes four main views, one to view notifications, one to view recipes, one to edit recipes, and one to view a map to both see dropped pins and create and edit geofences.

An example of a user editing a recipe from the user interaction component can be seen in Figure 3.2.

3.3 Activity Detection

The activity detection component (ADC) is the main module of RAO and is responsible for the actual detection of the user's physical state. In addition, the ADC will monitor the user's recipes, in order to see if the user's state transitions correspond to any existing recipe. If a recipe is triggered, the ADC will both take the appropriate action (defined by the **DO** of the recipe) and will create a notification which will be sent to the user interaction component in order to inform the user of the recipe being activated.

The ADC is also responsible for starting up and gathering information from other detection components as needed by the recipe's specified by the user. For instance, if the user has a recipe that relies on texts being sent and/or receiving, then the ADC must start up a component that will monitor SMS interaction.

The Android version of RAO implements the ADC as an IntentService class named TransitionIntentService. The TransitionIntentService class makes extensive use of Google's ActivityRecognitionApi, which provides periodic intervals to the TransitionIntentService as to the user's current physical state. As the ActivityRecognitionApi only uses low powered sensors, it was decided that this interval would be set to the smallest one possible.

3.4 Location Detection

The location detection component (LDC) was added as a standalone module as a result of the possible inaccuracy of using the ADC alone (see Section 4.2). The LDC's main job is to keep track of the user's whereabouts using GPS. The LDC interacts only with the ADC, and the LDC is activated and stopped by the ADC when required by the ADC. This is done so as to conserve energy, as the GPS is a high powered sensor. For instance, the ADC may activate location detection with high frequency during periods of possible transition in order to confirm physical state and location. At other times, location detection may be made less frequent or turned off entirely. The locations gather by the LDC is used for numerous tasks, such as possibly dropping pins on a map or creating and monitoring Geofences.

The Android version of RAO implements the location detection component as an IntentService class named InnerLocationService. The InnerLocationService makes use of Google's FusedLocationProviderApi, which allows applications to request location updates from Google. These requests occur at a programmatically defined frequency, which is controlled by the activity detection component.

3.5 General Detection

The general detection component (GDC) is responsible for all other smaller forms of sensing that may be required in order to determine user's state. For instance, the GDC may detect incoming SMS in order to fulfill certain user recipes. Other possible forms of general detection may be the detection of incoming or outgoing phone calls, incoming or outgoing emails, etc. All these additional detection components are only necessary for fulfilling specific recipes, whereas the ADC and LDC are required for fulfilling all recipes.

The Android version of RAO implements only one additional detection component due to lack of desire for more from small user surveys. Thus the GDC is currently implemented as one class called the SMSListener, as the GDC currently only detects incoming SMS. The SMSListener is started by the TransitionIntentService, and any information of incoming SMS messages is sent to the TransitionIntentService.

4. Evaluation

As it is in the nature of our application to be running all the time in the background and sometimes the foreground, energy consumption is of ultimate importance. However, to achieve the best accuracy for activity recognition, we found we had to increase the GPS polling rate to the highest frequency. GPS is of course a high powered sensor that draws a significant amount of power and rapidly decreases the lifetime of the phone's battery if left on all day. To examine how efficient or inefficient our application was, we investigated the trade-off between accuracy and battery draw by measuring the energy consumption of our app at different GPS intervals and recording how accurately it would recognize specific activities. We came up with two sample recipes and a set of five different tasks that were specifically designed to match with these recipes.

4.1 Energy

To be able to see how much energy RAO was consuming, we needed an energy profiling tool that would allow us to examine battery drain on a per-app basis. After experimenting with a few solutions like AT&T ARO, JouleUnit, and PowerTutor, we found that Qualcomm's Trepn was the most accurate and provided the most information. Trepn is a standalone application that runs on a smartphone and allows users to profile other applications and gather useful statistics on how applications use certain phone resources. We installed Trepn on an LG Eclipse 2 and a Nexus 6P and recorded the amount of battery consumed per millisecond to investigate exactly how battery usage corresponded to certain application events.

We created two recipes to simulate someone biking to class and locking their bike, and also walking to class and sitting down.

1. If biking, then walking, drop pin
2. If walking, then still, silence phone

Next, we performed five tasks specifically designed to hit, or match with these recipes thus forcing the testing phones to query GPS. These tasks were as follows:

1. Walk around in a circle for 1.5 minutes – screen on
2. Walk around in a circle for 1.5 minutes – screen off

3. Walk in a straight line for 1 minute, pause for 30 seconds, continue walking in a different direction for another minute – screen on
4. Walk in a straight line for 1 minute, pause for 30 seconds, continue walking in a different direction for another minute – screen on
5. Bike for 1 minute, walk for 30 seconds, stand still for 30 seconds – screen off

We performed each of the above tasks on both phones and with three different GPS states for a total of 15 tasks per phone. The three varying GPS states we tested were:

1. Only query GPS when instructed to drop a pin at a certain location
2. Query GPS every 20 seconds when detected activity is walking, running, or any other form of motion
3. Query GPS as fast as possible when detected activity is one of motion

Below we present a table showing the average and max battery consumption delta levels for a few of these tasks and configurations. The delta is measured in respect to a baseline battery consumption reading for this specific app. The GPS state in the table follows the numbering above.

Table 4.1 - Battery consumption of various GPS states, screen states, and phones.

#	Task	Screen	Phone	GPS	Max (uW)	Avg (uW)
1	Walk in a circle (1.5 min)	On	LG	1	165,920	107,137
2	Walk in a circle (1.5 min)	On	LG	2	433,242	187,376
3	Walk in a circle (1.5 min)	On	LG	3	441,614	284,704
4	Walk in a straight line, pause, walk (2 min)	Off	LG	2	9,664	1,372.63
5	Walk in a straight line, pause, walk (2 min)	On	LG	2	306,179	107,341.12
6	Walk in a straight line, pause, walk (2 min)	Off	LG	3	9,856	1,072.54
7	Walk in a straight line, pause, walk (2 min)	On	LG	3	450,093	300,041.47
8	Walk in a straight line, pause, walk (2 min)	On	Nexus	2	324,759	133,501.6
9	Walk in a straight line, pause, walk (2 min)	On	Nexus	3	466,467	367,005.21

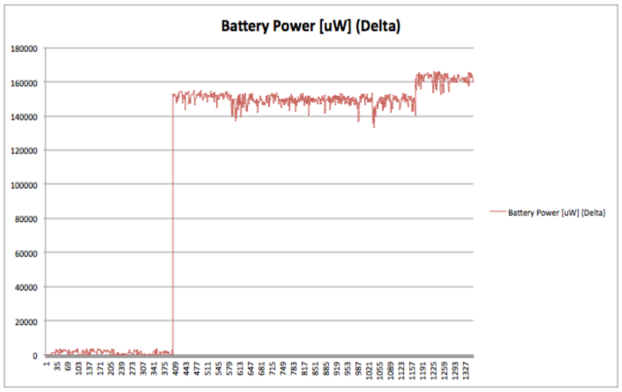

Figure 4.1 - Energy delta graph for item #1

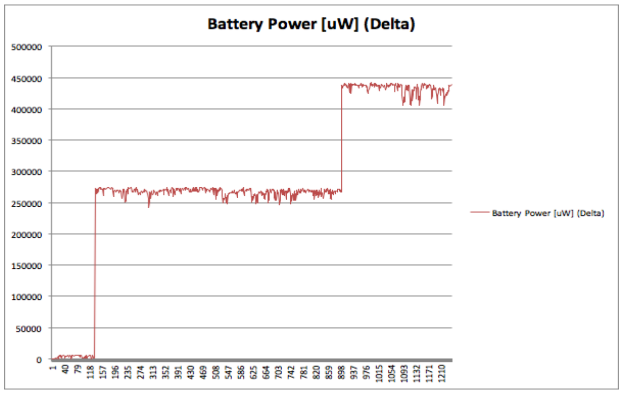

Figure 4.2 - Energy delta graph for item #3

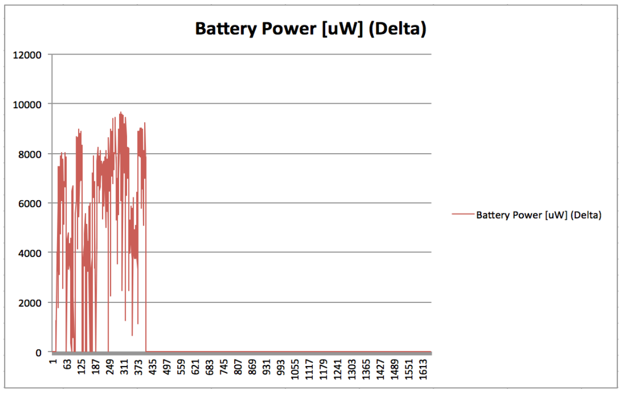

Figure 4.3 - Energy delta graph for item #4

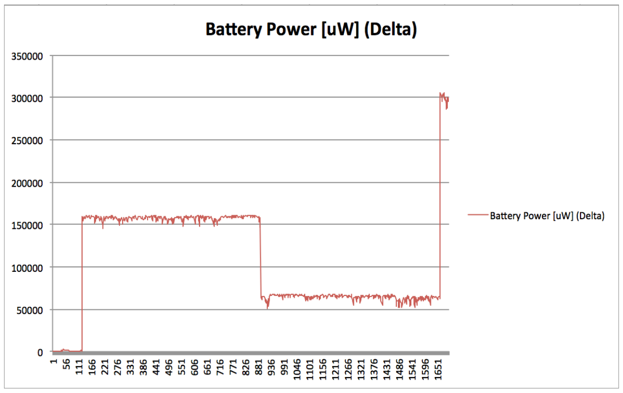

Figure 4.4 - Energy delta graph for item #5

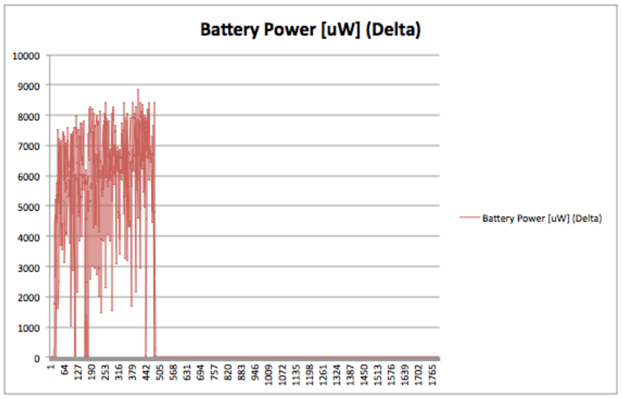

Figure 4.5 - Energy delta graph for item #6

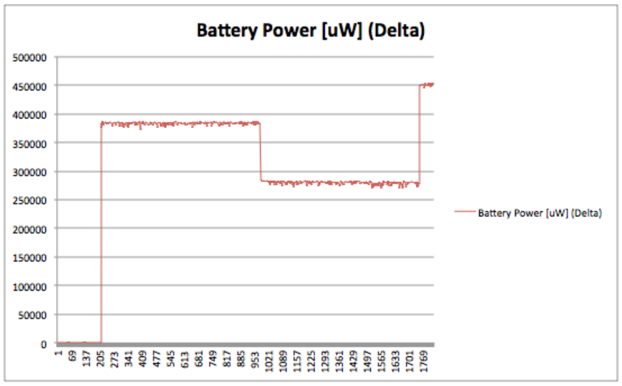

Figure 4.6 - Energy delta graph for item #7

In general, we found that having the display on is one of the largest causes for an app's battery consumption as can be seen when comparing items 4 and 5 from the Table 4.1 above. This confirms the result from previous studies such as [2]. The second largest consumer was GPS, as expected. When increasing the GPS use rate from once whenever a pin is dropped to polling as fast as possible for best accuracy, the average delta increase is 177,567 uW between items 1 and 3. In Figures 4.1 and 4.2 we see a delta of 0 uW for some time at the beginning of the test. We believe this is from the Google Activity Recognition API "warm-up" or startup time that we experienced. Often when we started our app, it took a little bit of time for it to start receiving updates and then would receive them frequently. The first sudden jump in both figures was when these activity updates start coming in, the second smaller jump in figure 4.1 was when a pin was dropped. The first jump in Figure 4.2 was due to activity updates and the rapid GPS polling. The second large jump in Figure 4.2 was because of the dropping pin in addition to continued GPS polling on the background thread. We're not too sure why it is such a large jump, but believe it might be because of GPS tail.

We faced some trouble with Trepn, especially with the Nexus 6P. Trepn would sometimes log spurious results similar to the ones shown in Figures 4.4 and 4.6 forcing us to often retest each task multiple times. On the Nexus phone, it wouldn't even receive battery updates as often as the specified millisecond intervals and would only report a few different battery levels. This would happen no matter how long we ran the test resulting in a graph that looked like a very general step function.

4.2 Accuracy

Since the Google Activity Recognition API only uses lower power sensor data from the gyroscope and accelerometer to predict activity, it is possible to fool these sensors into thinking the user is performing a certain action when they aren't. For example, if a user is sitting in class with their phone in their pocket and bouncing their leg up and down, the Activity API will think the user is currently walking. The same occurs when the person is just playing with their phone in their hands. We solved this issue by querying the GPS whenever the Activity API reports walking, or some other type of motion to check if the user is actually significantly changing location. If the GPS does in fact report movement, we know for sure that the user is walking, for example, and not just playing with their phone. To compare the activity prediction accuracy when just using the low power sensors to the accuracy when enhancing the prediction capability with GPS verification, we performed two tasks 15 times each with GPS on and then off. These results are shown in the following table:

Table 4.2 - Measuring the accuracy of how well a recipe is matched with actual actions

Task	**Accuracy**
If walking, then still, silence phone	93.3% (14/15)
If biking, then walking, drop pin	60% (9/15)

Table 4.3 - Testing accuracy with deliberate false positives

Task	**App says Activity = Walking (% incorrect)**
Sitting in class, bouncing leg w/o GPS	86.6% (13/15)
Sitting in class, bouncing leg w/ GPS	0% (0/15)

Clearly with GPS on, RAO is much more accurate and doesn't report any false positives. However, with this more accurate version of the app, it is more power intensive as shown in section 4.1. In a later version of the app, we would leave it up to the user to specify how frequently they would like the GPS to poll and, in effect, how accurate they require the application to be.

5. Contributions

The design of this application was sketched out by Aaron, with review conducted by Oliver and Roman. Programming at the beginning of the development process was done in a pair programming fashion in order to lay the basic skeleton of the code base so that everyone was clear as to the direction the application would take. This process included designing and implementing the recipe object. Once most of the essential framework code was out of the way, tasks were split up amongst the three team members. Aaron was responsible for implementing the main activity detection component, including designing and implementing its interaction with the user interface and all other detection components. Roman was given the task of implementing the location component along with any design specifications needed for interaction with the activity detection component, and with helping Oliver with the user interface. Oliver handled the general activity detection backend and recipe correlation and matching components as well implementing the user interface with help from Roman and specifications for interaction with the activity detection component set by Aaron.

In terms of finding a means of measuring energy consumption, all three members were equally active trying to find a viable solution. When one was eventually settled on, experimentation was conducted with the help of all three members. During experimentation, the typical setup was: Oliver would have both the testing phones in his pocket, Roman would hold the mobile hotspot and keep up with Oliver in order to main connection, and Aaron would carefully time each experiment and signal to Oliver and Roman when it was time to stop.

6. Conclusions

In this paper, we present RAO, an Android application that monitors human state transitions and lets users define the actions that should be performed whenever a specific state transition occurs. RAO utilizes both low and high-powered sensors to achieve the highest accuracy of transition detection. We find that without incorporating a high-powered sensor like GPS, the application can be fooled by some actions, leading to a 0% accuracy in those cases. Adding GPS increased accuracy by 86%. We find that Google Activity detection is quite accurate for most activities except for biking, which yielded a 60% detection accuracy. We conduct intensive energy profiling of RAO and confirm that the largest consumer of energy was the screen, with the second largest being the GPS. We also realize that software-based energy profiling tools for the Android OS are very hard to come by and often produce spurious results.

7. References

1. Incorporated, Q. T. (2013). WHEN MOBILE APPS USE TOO MUCH POWER A Developer Guide for Android App Performance, (December), 1–14.
2. Chen, X., Jindal, A., Hu, Y. C., & Chen, X. (2015). Smartphone Energy Drain in the Wild : Analysis and Implications Smartphone Energy Drain in the Wild : Analysis and Implications.
3. https://developers.google.com/google-apps/activity/