# Kai-Trace: A Means to Optimize Wireless Sensor Node Battery Life

Abhishek Sethi
Computer Science
University of Maryland, Baltimore County
abhishek9691@gmail.com

***Abstract*— Kai-Trace is a simple algorithm aimed at improving battery life of wireless sensor nodes through sensor control from the software level. As battery powered devices become more prevalent, a means to optimize battery life is necessary. There are many attempts at the hardware level, and a few at the software level. However, none take the approach of learning the node's behavior and dynamically adapting so that battery life is optimized on the fly. Kai-Trace aims to solve that issue by using a simple procedure to update average values of sensor on/off time, and toggling them depending on a specific threshold value. The information is saved in a B-tree like data structure that is organized from Day to Hour to Minute to Sensor from root to leaf. Kai-Trace learns and updates values to a specific day in an attempt to leverage one's habitual behavior. Though there are issues with consistency, and susceptibility to outliers, an Android implementation provided consistent results during experimentation. Kai-Trace does improve the battery life of an HTC One M7, increasing life by over 10%. Overall, Kai-Trace is successful in improving cellphone battery life, and can be expanded to general wireless sensor nodes.**

***Keywords—energy, arithmetic mean, gossip algorithm, battery optimization, Android, wireless sensor nodes, B-tree***

## I. Introduction

### A. *Basic Wireless Sensor Node Architecture*

Wireless sensors are used in the world around us for any purpose imaginable. Their uses vary from underwater monitor networks to tracking enemy movement in the harsh deserts. The applications of wireless sensors are endless, with new ideas surfacing everyday as the need demands it. Most applications are inconspicuous, usually hiding the physical sensor itself in order to protect the node itself, but some are more obvious, such as robotics or cars.

Wireless sensors are ubiquitous, but what exactly are they? A basic definition of a wireless sensor is a sensor that passively monitors events in its surroundings, processing the signal and passing them onwards to other nodes within a network. Figure 1 displays the basic architecture of a single wireless sensor node.

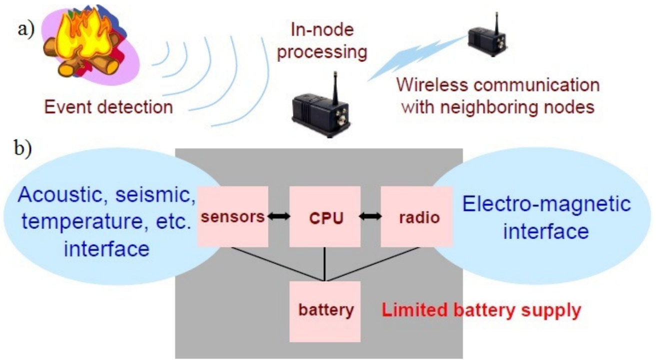

a)
Event detection
In-node processing
Wireless communication with neighboring nodes
b)
Acoustic, seismic, temperature, etc. interface
sensors
CPU
radio
Electro-magnetic interface
battery
Limited battery supply

Figure 1: a) A general overview of the general use and process from event detection to information propagation of the signal processed by a single node. b) The general architecture of a single wireless sensor node. [7].

As shown, there are a few main components of a wireless sensor node. Figure 1b shows the general architecture and circuitry involved with a wireless sensor node. In the circuitry, we can find the sensor, CPU, and radio. The sensor can monitor and passively survey its surroundings, taking signals and passing them to the CPU for processing, or to a signal processor. The CPU then performs computation or relays the information from the sensor to the radio for transmission. The CPU also controls the usage of each sensor, and when they should be toggled on/off. If needed, the radio is used to send and receive information from other nodes. This allows a single wireless sensor node to communicate with other nodes, allowing a network to form. The entire process is controlled by the CPU, and more importantly, powered by the battery. The battery is generally a rechargeable battery to avoid the need for constant battery replacements, especially if the nodes are placed in harsh or remote environments. The entire circuitry is enclosed in an encasing that is suitable for the nodes purpose and placement.

Figure 1a shows the usage and purpose of wireless sensor nodes. Once an event triggers the sensor on the node, the CPU wakes up and begins processing the signal, and relaying the information if need be. Using the radio, nodes are able to transmit information to their peers via various methods. The benefit to this is to avoid the need for long wires to physically connect the sensor nodes, offering a level of portability and area coverage. This creates another set of problems, but that is for another paper.

### B. *Wireless Sensor Node Issues*

By looking at the basic architecture of a sensor node, it is obvious that a range of difficulties arise when trying to deploy and use sensor nodes. The most obvious is placement. Since sensors and radio antennas have limited range, the sensors must be deployed in such a way that area coverage is maximized, while using the smallest amount of sensors and maintaining inter-node connectivity. Nodes should be able to locate their peers, and identify their own location relative to their peers. Using a strong radio antenna to increase area coverage of nodes in a network is plausible, but then that directly increases battery usage. There are many methods and algorithms to determine optimal spacing and placement that account for coverage and battery usage, including random deployment, pre-planned deployment, and various other methods. This is currently a hot area of research, and will continue to advance as time goes on.

Another problem is mobility. The inter-node communication was made wireless to steer clear of having physical connections using wires. However, what will happen if a node is out of range from the current node due to unforeseen circumstances, such as natural causes. A simple solution is to give the node the ability to relocate using some form of movement (motors, wheels, etc). This is an excellent solution, but the components required to move the node may require a large amount of battery power, draining battery quicker.

Another issue is processing capabilities. A large number of tradeoffs must be considered here. There are small processors capable of performing intense computations that would physically fit well on a node, but then those processors generate large amounts of heat, and require large amounts of power. The generated heat can potentially harm other onboard components, causing the drawbacks to outweigh the benefits. Therefore a tradeoff must be made, and nodes must be equipped with processors that are sufficient enough to perform the necessary tasks. In general when designing a node, the amount of processing required is reduced so the processor does not generate too much heat, or deplete the battery quickly.

Overall, there are quite a few issues with wireless sensor nodes, but there is one common theme amongst these issues. The wireless sensor node capabilities are limited by the battery capacity available. Once the battery dies, the entire node is rendered useless. There are many research efforts aimed at improving battery life, ranging from improvements on a battery's chemical capabilities, to software optimization to minimize a components energy footprint. A common technique is to use the node's custom software to turn off the various sensors/radios on a node to reduce the amount of wasted battery power. This has been a proven method for improving battery life, as the energy required turning off a radio/sensor is far less than the energy lost having the sensor/radio constantly on. For example, a common technique to control radios and sensors would be to shutdown the sensor circuitry via the main CPU. The power required to shutdown the sensor would be Pdown, and the power required to start the sensor is Pup. The power saved from the sensor being off is Psaved. In general, Psaved is significantly greater than Pdown, making it worthwhile to shutdown the sensor/radio for battery optimization purposes. Usually, the processor is also throttled to use less power, saving more battery. This is generally a good process, but the limitation is that if the sensor/radio is off and misses an event or radio transmission, then that could arise in problems. Occasionally, a processor has predetermined sleep times for each sensor/radio, and can toggle the sensors/radios based on this timing. This results in less transmissions missed, but the issue here is the times being predetermined. An evolution to this idea would be to have the processor automatically learn when and for how long to shut off the sensors.

### C. *Kai-Trace Goals*

The goal of this project is to improve battery life of a node via nodes software capabilities by learning and automating, in real time, the control of sensors/radios antennas as opposed to hardcoding values into software. This will ideally be built into the software as a lightweight process, in order to minimize the processes energy footprint as well. The ultimate goal is that the Psaved outweighs Pprocess (power used by the process performing the energy conservation).By performing real time learning, the process will prevent radios from draining battery life by remaining on, and allow the system to adapt to real time usage of the sensor node. One assumption that is made is that in general, the Pdown and Pup are miniscule compared to shutting off a sensor/radio for a given amount of time. The project is conveniently named Kai-Trace, as the idea is to trace the usage of all sensors in order to optimize energy conservation.

Kai-Trace is a proof of concept approach to optimizing the battery usage of a node through software. In order to provide an implementation, a prevalent wireless sensor node will be used. This node is the common smartphone. Based on the definition of a wireless sensor node, and the architecture provided in Figure 1, a cellphone can be classified as a type of wireless sensor node, and will be treated as such. A cellphone contains radio antennas and sensors, such as a WiFi antenna, accelerometer, etc. This gives the smartphone the ability to communicate with other smartphones, large cellphone towers, and wireless internet hotspots. As all wireless sensors, cellphones suffer from the same problems as a typical wireless sensor node, but primarily from limited battery life. For example, iPhones have been reported to be able to have a 10 hour battery life whilst using video and internet, but common sources report less than 6 hours of life instead [2]. The reason a cellphone was chosen as the wireless sensor device is the ubiquity of cellphones, the potential impact of this approach on improving the lives of cellphone users, and how readily available the development tools for cellphones are. There have been some improvements on battery life, but they have not improved dramatically as processing capabilities of cellphones are rapidly increasing as well. Overall, a push for better battery life would not only improve the use of cellphones, but also the lives of consumers using cellphones.

In order to provide a proof of concept, the general procedure will be implemented for the Android operating system. The reason for this choice is that Android devices are notorious for having large batteries, but sub-par battery life compared to their iPhone counterparts. Also, since Android applications are built using Java, this will speed up the development process as the authors have prior experience with Java development. Kai-Trace achieves its goals by collecting information about when each radio/sensor is on at a specific time of day. Once knowing this, simple metrics are determined to aggregate this data from multiple days. By manipulating the onboard sensors and radios, the end result should be improved battery life, taking another step forward in terms of smart battery optimizers.

### D. *Related Work*

On the Google Play Store, there are hundreds of applications on Android that are aimed at improving battery life. Most of them require a user level configuration, involving the user with the battery optimization. This is nice, but the goal is for smart, automated battery savers. There are a few out there deemed as smart battery savers. One is the Smart Battery Saver by Bitdefender [3]. This application is smart in the sense that it toggles various radios and sensors based on the screen being on/off, and if the phone is locked or not. These are great features, but do not give the sense of the application being able to learn on its own, and be smart in that sense. Another app that claims to be a smart battery saver is the Smart Battery Saver & Booster by Muddy Apps and Games [6]. This application is great, but again requires user involvement in order to optimize the battery life, configure settings, or tell the application to kill processes or toggle sensors. Once again, this is smart in the sense it uses predetermined methods and values to perform battery optimization, but does not learn how the user operates their cellphone, and adjust to that. Overall, there are applications that claim to be smart, but all require user involvement in optimization as opposed to the application learning on its own.

## II. Design

### A. *Core Algorithm*

The process that controls all the sensors, and performs computation to determine which ones to toggle is broken into two pieces. One is the underlying data representation, while the other is the algorithm used to perform the computation and update. Both are discussed in further detail below.

### B. *Data Representation*

There are multiple ways that the data collected by Kai-Trace can be logically represented in. The most obvious would be to simply store everything in an array. Each entry in the array would be a single time stamp with sensor information collected throughout the day. Though this is a simple approach, there are a few issues with it. One is this representation does not accommodate time ranges, and relies on single timestamps. Each timestamp will be different, leading to two timestamps that are seconds apart to be stored as 2 different entries in the array. It would be simple to account for time ranges by doing a simple if statement check or computation prior to this, but this is extra work that needs to be done, and will result in more computation needed for the process.

After going through multiple options, it was decided to use a data structure similar to a B-tree. By going with a B-tree like structure, it allows us to have sorted keys within each node, making tree traversal easier. Also, traversal down a tree is efficient since the maximum height of a B-tree is generally small, less than 5 levels. The nature of how the children are organized also allows us to accommodate time ranges as opposed to specific timestamps. Though some pre-checking similar to the array variation is done, overall, the ability to search in less than O(log(n)) time makes the B-tree advantageous, as opposed to a worst case O(n) search time for an array. Overall, the performance and structure of a B-tree make it suitable for data storage purposes in Kai-Trace.

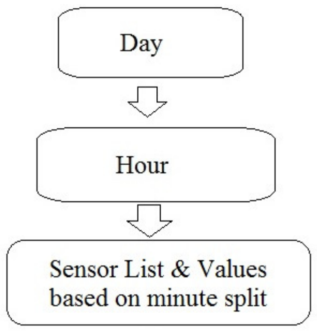


Figure 2: Data hierarchy and levels of the B-tree for saving and updating the values for the sensor list.

Figure 2 provides the general hierarchy for each level in the data structure. To describe the B-tree elements of this hierarchy, we shall begin with the day node. This is the root node, where all data structure operations will begin. This node has seven keys representing the day of the week, each for Sunday through Saturday. The day node will have seven children; each one being an hour node. Each hour node will contain 24 keys, ranging from 0 to 23 to represent the hour of the day in 24-hour format. Now, each key in the hour node will have only 3 children. These children are the minute nodes containing sensor information. The final algorithm executes every 20 minutes, which is one third of an hour. Since the minute nodes must all cover the hour, 3 nodes are used, resulting in a left, middle, and right child, covering minutes 0-19, 20-39, and 40-59, respectively. This rule is to be hardcoded into the software. This provides full coverage of a timestamp. The reason for going with 20 minutes is that it provides accuracy in determining sensor on time, without sacrificing too much space. Using a larger value would result in inaccurate values, but less space used, whereas the opposite would result in higher accuracy, but the process would execute more frequently and use more space, leading to more battery being consumed during each execution. Each of these minute nodes simply contains an array of doubles that represent the various values for determining each sensor's power state. These doubles correspond to the sensors/radios that can be controlled. This is elaborated in the section of Data Aggregation.

It is worth noting that timestamps are being used here. Generally, timestamps are an unreliable source of time measurement due to a multitude of reasons, but assuming the cellphone is deemed an isolated node, and some clock skew is tolerable, we can use timestamps for simplicity purposes. It is worth noting that there are a total of only 3 levels in the logical data representation. This makes searching, and traversal easy if need be. There are only a total of 7*24*3 = 504 nodes in the entire data structure at any given time. As for the space requirement, the first two levels contain integer arrays, which are a total of 4 bytes per integer, and 8 bytes for the pointer (assuming basic C variable sizing). This gives a size requirement of 36 bytes for the day node (7 values at 4 bytes each, plus the initial pointer to the root node), 104 bytes for each hour node (24 values at 4 bytes each, plus the pointer to the node), and 56 bytes for each minute node (6 values at 8 bytes each, plus the 8 byte pointer to it). Overall, there is a total of 4,796 bytes (4.8 KB) used in memory. This is a relatively small memory footprint, especially when modern smartphones contain over 1 GB of RAM for processes, making the B-tree style approach an ideal one for representing this data, and performing operations over it.

### C. *Data Aggregation Algorithm*

Having the data structure set is nice to have, but now the data structure must be put to use. In order to carry out the battery optimization, a specific algorithm was created to update and modify sensor values so that sensors can be turned on/off as need be. This operation is carried out every 20 minutes, as to much the minute values of the data structure presented. The pseudocode is presented in Figure 3.

```
//A global variable referencing the data structure discussed previously. This should be available //throughout the
duration of the program
current_sensor_values = B_Tree_Data_Structure

update_values():
    //an array of integers (only 0 or 1), all initialized to 0 to begin with.
    new_sensor_values <- [ 0, 0, 0, 0, 0, 0]
    threshold = 0.35 //the threshold value stating whether to turn a sensor on or not
    for each sensor $i$ onboard: //sensor $i$ corresponds to the $i^{th}$ element of new_sensor_values
        if( $i$.is_on )
            new_sensor_values[i] = 1
        end
    end
    //saves the returned, updated sensor values
    updated_values <- current_sensor_values.add(new_sensor_values)
    for i from 0 to size(updated_values)
        if( updated_values[i] > threshold)
            toggle the $i^{th}$ sensor to on
        else
            toggle the $i^{th}$ sensor to off
        end
    end
end

//in the Data Structure
B_Tree_Data_Structure.add(new_sensor_values):
    curr_day <- get_current_day() //some method to get the current day
    curr_hr <- get_current_hr() //some method to get the current hour
    curr_min <- get_current_min() //some method to get the current minute
    //the method to retrieve the currently set sensor values for the current timestamp
    curr_values <- day.get(curr_day).get(curr_hr).get(curr_min)
    for i from 0 to size(curr_values) //the size can be hardcoded to 6 to represent each sensor
        //compute the arithmetic mean
        new_value = (curr_values[i] + new_sensor_values[i])
        curr_values[i] = new_value
    end
    //sets the updated values in the data structure
    day.get(curr_day).get(curr_hr).get(curr_min).set(curr_values)
    //returns the newly updated values
    return curr_values
end
```

Figure 3: Pseudocode explaining each step of the algorithm for adding to the data structure, and for how to toggle sensors accordingly, and what is computed at each point.

The algorithm to update and manipulate the values is fairly straightforward and simple. The goal is to have a small footprint, and be quick to execute in order to minimize battery consumption by the Kai-Trace application. As noted in the algorithm, we first begin by determining whether or not a sensor is on or not. This is represented by a 1 or a 0, indicating on or off respectively. There are two exceptions. One is for screen brightness, which is a value between 0 and 255, indicating light intensity for the screen, and the other is the phone's ringer intensity, which is either silent, vibrate, or normal. This was chosen for simplicity's sake, and can be improved upon in future work. Once this information is collected, it is passed on to be inserted to the appropriate location of the data structure. This is determined based on the current timestamp collected by the process itself. The timestamp is then used to obtain the array of sensor values. Here, the algorithm performs computation to determine the newly updated value.

In order to aggregate the new values with the current values, an arithmetic mean is computed between the two. This is done by simply retrieving the values, and saving the result in the current values array. Though the arithmetic mean seems naïve, the idea was borrowed from the Gossip style algorithms [4]. These algorithms are commonly used in networking, or distributed systems, where the nodes need to communicate information with each other. The general process is that a node will randomly select a neighbor it is connected to, and exchange some information with them, such as a number. The node with then have 2 values, its own value and its neighbors value. The current node will now compute some metric, such as an arithmetic mean between the two values, and then save this result as the node's new value. The node then picks another random neighbor, and repeats the communication procedure. Of course, the value will change each time, but overtime, the value will then converge to a single value that is common to all nodes in a network. The same idea applies here. Each array of values is treated as a node. The nodes communicate, aggregate their information, and then finally save the value. Over time, the idea is that certain sensors will be on at certain times, resulting in the arithmetic mean being closer to 1 during that specific time range in the data structure. Ideally, the values will converge to some number between 0 and 1 to indicate the power state of the sensor. The gossip style idea for data aggregation fits nicely with the idea of a lightweight process.

Once computation is done, the updated values are returned, and then used to determine what power state a sensor should be in. This is dependent on a threshold value set by the user. It is currently set to 0.35, but can be any value. For the purposes of brightness and sound profile, the threshold is irrelevant as the value can be greater than 1 for both sensor cases. Currently, we are experimenting with this value to see what works best for the application. Note, that there are multiple for loops in figure 3, but these loops iterate over arrays of size 6. This provides a hardcoded, small and constant value, making iteration almost negligible in the grander scope of the process.

## *D. Android Implementation*

The algorithm itself is depicted in Figure 3, but the implementation provides greater insight to enhancements and optimizations made at a program level. The implementation was done for the Android operating system. There are a couple of terms that must be defined before jumping into the implementation itself. Figure 4 provides a high level overview of the different parts of an Android application and their interaction.

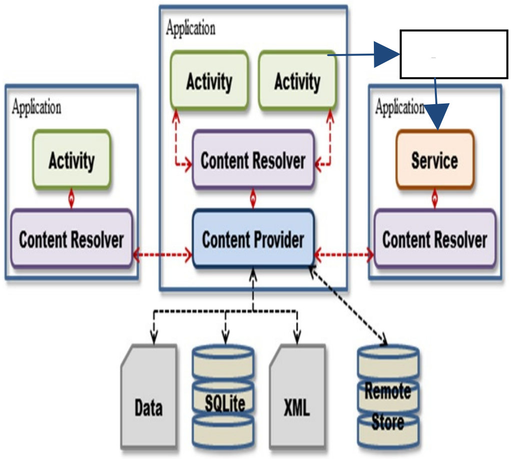
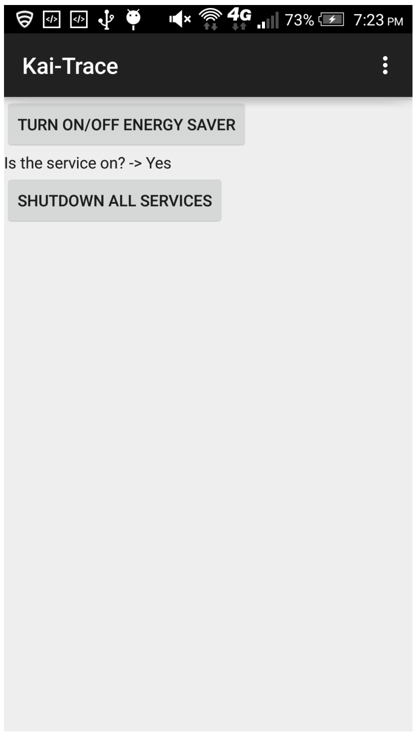


Figure 4: An overview of basic application components and how they communicate [5].

First is an Activity. An Activity in an Android application is similar to a web page. It contains the part of the user interface the user interacts with, and is the portion of the application that will offer the user control over the application [1]. Next is an Intent (the red arrows in Figure 4). As with switching web pages, switching between Activities requires some sort of action. An Intent is such an action. It allows for communication between Activities, and for information passing as well. Next is an Alarm. An alarm is simply a class in Android that allows the programmer to specify when to regularly execute a specific action. An Alarm can be executed at a specific time or even after a specific time limit. An Alarm then calls a Service to repeatedly execute when the Alarm goes off. A service is essentially a background task for Android. Multithreading is possible, but Services are safer features, and offer more direct interaction with parts of an Android application. Finally are the Content Resolvers and Providers, which help to control content delivery between storage areas and Activities.

Now, as far as the activity in the implementation, there is only one activity. Figure 5 shows the first and only activity that the users see. This activity was made to provide a minimal user interface for the user to configure since the bulk of the processing and work is done in the service portion. The activity simply provides information on if the background service is on or not, and the ability to minimize all possible sensors through the Shutdown All Services button. Alarm and service classes are created to perform the background processing necessary to update the sensor values. The alarm class is set to execute the Service class, which performs the update procedure, every 20 minutes from the start of the application/service. This is crucial to the effectiveness of the application. Note that the background service can be toggled as well.



Is the service on? -> Yes

Figure 5: Screenshot of the initial screen upon opening the application. This was taken on the HTC One M7 test cellphone.

Along with the entire alarm and services class, the important part is how the data structure is implemented. Performance is crucial to the use of this data structure. For performance purposes, the data structure is implemented using a HashMap to establish the Day to Hour relationship, and the Hour to Minute relationship. A HashMap is used because the map component makes it easy to maintain key-value pairs, and the hashing allows for constant time access to each component necessary. Hardcoded array key-values could also have been used, but the improvement is marginal in comparison to using a HashMap directly. As for the sensor list, rather than create 3 separate nodes, the Hour node now has 1 child per key. This child is a wrapper class called HourSplit. This class encompasses an entire hour, and maintains 3 different lists for each of the minute ranges specified in the Data Representation section. The 3 lists are simply ArrayLists. This offers the performance of an array in a Java data structure that can be easily manipulated. This can be interchangeable with a standard array if more performance is required, but that was not done here. The goal is to decrease the execution time of the service by making as many data structure accesses constant time if possible, which is why an emphasis on arrays and hashing is placed here.

As another optimization, array indices were pre-determined to correspond with specific sensors and values. For example, index 0 in the minute array correlates to the WiFi control, and 1 is Bluetooth, and so on. This hardcoding allowed for small optimizations, as opposed to searching through the arrays and performing comparisons to check which sensor is currently being analyzed. Also, in order to optimize the computation of each sensor arithmetic mean in the arrays, loop unrolling was performed as the numbers of iterations were previously known. Ideally, this will save time and space in the final assembly code and execution on the cellphone. Overall, the final implementation was successful, providing decent results during the testing phase.

## III. Results

### A. Hardware & Software

The Android version used is Lollipop 5.0.2, customized to have HTC Sense 6.0. It is worth noting that in the newest version of Android, Google removed programmatic access to mobile data and GPS sensors without user consent. This restricted the sensors and features I could control automatically to WiFi, Bluetooth, Auto-Sync, screen brightness, sound profile, and the screen being on/off. The cellphone used for testing is an HTC One M7. This cellphone contains a 2300 mAh battery. The phone was released in 2013, making it relatively old technology, hence the smaller battery. Generally the phone provides around 10 hours of battery life before requiring a charge (drops to approximately 10% of battery life).

### B. Experimental Results

Unfortunately, there was not enough time for thorough testing of the application. There were some preliminary results obtained, resulting in interesting results. Whether or not the results are significant is currently up for debate. The experiment was conducted by attempting to do the same activities everyday during the same time periods to ensure consistency across days. During certain hours, the phone would lay dormant with nothing on, and at other times radios would be on/off during certain times to ensure battery is used, and the application will learn when to turn on/off the sensors. Data was collected over a 10 hour time period, and simply the battery percentage was read off my phone screen.

A slight modification was made to the data structure. Due to the time constraints, the day node was reduced to only 1 key. Normally, there are 7 days, and the timestamp with the day determines where to store the updated sensor values. By reducing it to 1 key, we treated all days as the same, so we were constantly updating the values everyday, as opposed to only on specific days as proposed. This allowed us to simulate using the application for 5 weeks and one day’s values are updated. The issue here is that consistency is the key to success. The theory is that the user has a specific routine they follow, and this application should be able to learn this routine as well, optimizing battery life along the way. As long as the everyday activities do not vary much everyday, the experiment had potential to create useful results. Fortunately, this was the case except for Thursday. Figure 6 and Table 1 show the results of this experiment.

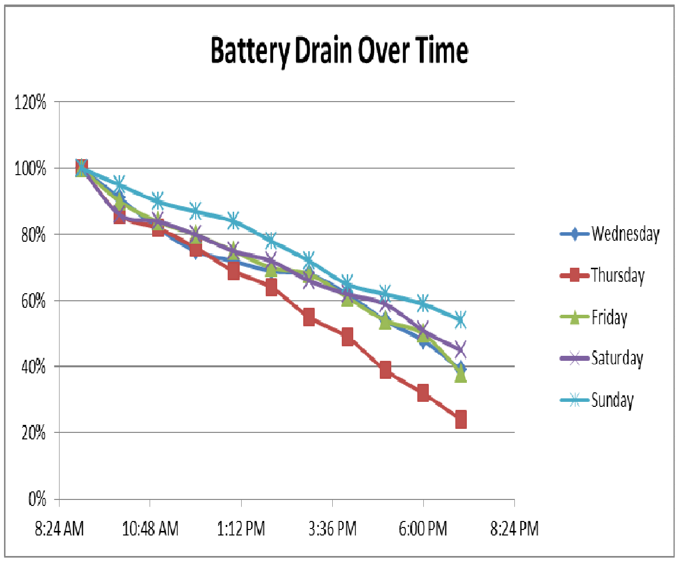


Figure 6: The battery percentage over time collected over a time period of 5 days.

| Day of the Week | Wednesday | Thursday | Friday | Saturday | Sunday |
|---|---|---|---|---|---|
| Total Battery lost (%) | 61 | 76 | 62 | 55 | 46 |

Table 1: Cumulative battery loss over each day as time went on. Note that the first 2 days were training essentially, and the last 3 were testing days. The smaller the value the better result as less battery is drained.

| Rate of battery drain per day for each hour (battery % / hour) | | | | | |
|---|---|---|---|---|---|
| | Day of the Week | | | | |
| Time of Day | Wednesday | Thursday | Friday | Saturday | Sunday |
| 9:00 AM | 9% | 14% | 10% | 14% | 5% |
| 10:00 AM | 9% | 4% | 6% | 2% | 5% |
| 11:00 AM | 7% | 6% | 4% | 4% | 3% |
| 12:00 PM | 3% | 7% | 5% | 5% | 3% |
| 1:00 PM | 3% | 5% | 5% | 3% | 6% |
| 2:00 PM | 1% | 9% | 2% | 6% | 6% |
| 3:00 PM | 6% | 6% | 7% | 4% | 7% |
| 4:00 PM | 8% | 10% | 7% | 3% | 3% |
| 5:00 PM | 6% | 7% | 4% | 8% | 3% |
| 6:00 PM | 9% | 8% | 12% | 6% | 5% |

Table 2: This table displays the battery drain per hour. The slope between each hour was calculated and presented here. Values were made into absolute values to help with understanding.

## IV. Discussion

The experiment yielded useful results. Figure 6 shows a time course of the change in battery drain over time. It is important to note that Wednesday and Thursday were training days. After these days, the application began behaving and showing improvements in battery life. Table 2 shows the rate at which the battery decreased per hour. This was presented to indicate the consistency in the experimental procedure, as an attempt to ensure consistent actions and results. As seen in the first three days, specifically Thursday and Friday, there was consistency in the replication of the experimental protocol since battery drainage was similar on those days. However, Thursday seems to be an outlier; hence it has a dramatic decrease in battery in comparison to all the other days. This caused the learning portion of the application to revert backwards, and unlearn what it had already learned. This created tons of false positives that day, resulting in a regression from the learning of the day before by readjusting the sensor parameter values. However, once the application began learning and controlling the sensors, battery life started to slowly improve. As seen in Figure 6, on Saturday and Sunday, the battery percentage remaining creeped higher in comparison to the battery percentage remaining on the other days. This is a sign that controlling the radios/sensors in fact improved battery life.

Table 1 also shows the general trend of battery improvement. The total battery loss after the 10 hour period began to decrease on Saturday and Sunday. This is an indication of the application's ability to learn and modify the cellphones settings to optimize battery life. Unfortunately, since this is only a single sample run, statistical analysis could not be performed on the data. This opens up the possibility that the results were skewed due to other factors, such as battery age, phone age, battery health, etc. Once more data is collected, it can be confirmed whether this application can ensure battery optimization.

Though the application produced useful results, there were quite a few issues with the approach, both theoretically and experimentally. The first was that the process itself was not timed. It is possible that the process used CPU bursts to maximize runtime and CPU time, or it is possible the process was held back and did not execute in a short time frame. Along with monitoring the process, more hardcoding could have been done. For example, the HashMap and some computational numbers are determined dynamically, such as sizes of intermediate arrays. If this value were hardcoded, this could be a small optimization, but potentially lessen the CPU usage of the process. Also, decreasing local variable usage may potentially have a certain speedup that could be noticeable.

Another issue is the lack of data. Due to the nature of the gossip algorithm, it takes a large number of nodes or iterations before reaching convergence on the arithmetic mean. This experiment used only 5 days, and amongst those was one inconsistent day. 5 days seems to provide enough results for a proof of concept, but it does not provide enough information on whether the application can sustain this learning for long. More data on whether the arithmetic mean in fact converges would be necessary to improve upon this study. Another improvement could be to use some other metric besides the battery percentage indicator. The indicator is fairly accurate, but it also rapidly changes or is sometimes inaccurate due to the age of the HTC One M7. Some other metric should be used to determine battery drainage other than the phone's built in indicator.

The space used by the data structure is also an area for improvement. It was calculated to be about 4.8KB, but that is the data structure alone. Since HashMap and ArrayList implementations were used, it is possible the data structure and process use up a larger chunk of memory, potentially twice as much. 9.6 KB may still be relatively low for memory usage on a modern cellphone, but if this approach is to be applied to other wireless sensor nodes, then memory usage should be taken into consideration.

One gaping issue with the application is its susceptibility to outliers. As seen in Table 1 and Figure 6, when Thursday was performed out of sync with the remaining days, there were issues with the learning procedure. Suddenly the values were not as accurate, and it were as if the application regressed in its learning. Logically and mathematically this makes sense. A perfect example would be if the sensor value is consistently a 1, but one day the sensor was off, so now the new mean is (1+1)/2 = 0.5. If the sensor is off 1 more time, then the new mean is now 0.25, which is below the specified threshold value in Figure 3, causing the application to predict the sensor should now be shut off. Only after 2 outliers, the sensor is now predicted to be off as opposed to on. This causes fluctuation amongst the values. There is a benefit here, as such fluctuation may be normal or the user's habits have changed (tends to turn off sensors now instead of on), forcing the application to react to these changes immediately. An improvement could be to use some other metric to predict the power state of the sensor, such as a weighted mean. However, this should only be done if the arithmetic mean has been proven to not converge after weeks of data, not days.

## V. Conclusion

Overall, there are quite a few gaping issues with the implementation, but it has proven to be successful thus far. By utilizing hashes and array accesses, the B-tree like data structure achieves a good level of performance compared to other solutions. Though the arithmetic mean is susceptible to outliers, there is a silver lining to its usefulness in adapting to changes quickly. Fortunately, the mean is easy to compute, reducing the computational work required by the Service responding to the alarm in the Android system. Not only can this approach be applied to cellphones, but it can also be applied to other wireless sensor nodes found in any location, especially in high network traffic areas. With some tweaks to the space and computational requirements, this approach has the potential to improve the battery life of cellphones, and all wireless sensor nodes.

## References

[1] Activity. http://developer.android.com/reference/android/app/Activity.html. 15 Dec 2015.

[2] Carnoy, David. One spec Apple didn't improve in iPhone 6S: Battery Life. http://www.cnet.com/news/iphone-6s-battery-life/. 10 Sep 2015.

[3] Hindy, Joe. Smart Battery Saver App Review. http://www.androidauthority.com/smart-battery-saver-review-317704/. 17 Nov 2013.

[4] Hoff, Todd. Using Gossip Protocols For Failure Detection, Monitoring, Messaging And Other Good Things. http://highscalability.com/blog/2011/11/14/using-gossip-protocols-for-failure-detection-monitoring-mess.html. 14 Nov 2011.

[5] Mobiminds. Android Project Structure And Activity Life Cycle. https://bestandroidtraining.wordpress.com/2013/03/30/android-project-structure/. 20 Mar 2013.

[6] Muddy Apps and Games. Smart Battery Saver & Booster. https://play.google.com/store/apps/details?id=com.muddyapps.Smart.Battery.Doctor&hl=en. 31 Jul 2015.

[7] Younis, Mohamed. CMSC684, Spring 2015. Lecture 2.