EnergyAware: Personal Energy Monitoring System

Nishad Gothoskar
Carnegie Mellon University
ngothosk@andrew.cmu.edu

ABSTRACT

CCS Concepts

•**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; •**Networks** → Network reliability;

Keywords

ACM proceedings; LaTeX; text tagging

1. INTRODUCTION

Many times in the fields of Smart Buildings and Smart Homes, we disregard the essential concept of Energy Proportionality. In commercial buildings and residential homes, it may be true that there is a high inherent overhead and therefore proportionality cannot be achieved. But on a smaller scale, like apartments and single bedrooms that most young people live in, this proportionality is truly achievable, given that there is much less energy overhead. When we are not at home, our energy usage should be close to nothing and it should increase when we are present. But, the fact that our energy usage is not proportional should not be blamed on the users themselves. In fact many users would be readily willing to reduce their waste usage and adjust their overall usage. But power and energy usage measurement and statistics are not readily available or accessible to users, so they have no way of knowing how or in what ways they should change. In this paper we look to change this so users can be more aware of their energy consumption.

1.1 Jevon's Paradox

Jevons paradox says that when the efficiency of usage of a resource is increased there will be an increase in usage and demand and in effect there will be little to no change. While this may not be completely true when it comes to energy resources, it may be limiting our potential of energy savings. There is an aspect to Green Computing that cant be solved by technological innovations alone. In some ways, significant improvement is dependent on changing user's usage and more importantly their demand habits.

WOODSTOCK '97 El Paso, Texas USA
© 2016 ACM. ISBN 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

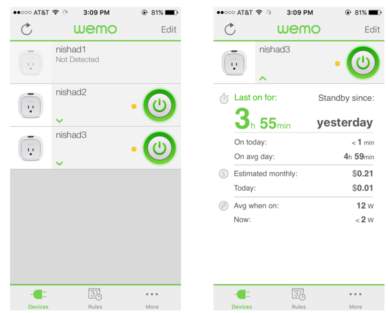

Figure 1: Screenshots from Belkin Wemo iOS App

1.2 Belkin Wemo

Our measurement sensors are Belkin Wemo Insight Switches. These switches come with free software through an iOS and Android application. The application provides the ability to see minimal statistics about each plug and also control it remotely as shown in Figure 1. It is a quite thorough and useful application in theory, but practically it is hard to use and does not provide a solution to the problem we are attempting to solve. In this project we are not looking to remake or replace their existing application. Instead we are trying to extend and implement another aspect of energy management that the Belkin application was not meant for.

2. HARDWARE

In the deployment of the EnergyAware system, we were conscious of the fact that it had to be easily deployable. Complex hardware setup would be unattractive to users and therefore the "market" for the system would be extremely limited. As mentioned above, the measurement is done by Belkin Wemo Insight Switches. These are WiFi connected devices which can be accessed by devices on the same WiFi network. The cost of these are around $50 dollars each, which is on the more expensive side. However, we don't need these devices exactly since they have additional features that our system does not use which add to the cost. However sites like `openenergymonitor.org` have designs that can be made for much cheaper and would be sufficient for the EnergyAware system. Now, the second piece of hardware is the center of system, which runs all the main EnergyAware code. In our case, we ran it on an Intel Edison Breakout

Table 1: Powering Intel Edison, Yocto 3.0

Activity	Power (mW)	Voltage	Current (mA)
Booting Up	944	4.97	190
Using Wifi	803	5.02	160
Idle	455	5.05	90
Powered off	<51	5.12	<0.01

board, however it could be run on any system that will constantly be awake. Since it would be inefficient for someone to leave a desktop or laptop in there home on at all times, an Edison Board or any equivalent system like a Rasberry Pi would work well with little overhead.

2.1 Power Ratings

It is clear that we must ensure that our added hardware doesn't itself significantly increase energy consumption. Our system would be useless if we added a large energy overhead without significant savings. The Wemo Switch has a power usage of less than 1.5 Watts. Table 1 shows the ratings of the Intel Edison. It is clear that the power overhead of running EnergyAware is minimal and that adds to the inspiration of the system being something that can't hurt but only help improve habits and reduce wastage.

3. ARCHITECTURE

The layout of the entire EnergyAware system stresses reliability and dependability. But, it also stresses usability given that this is a user-centric system and if the user does not enjoy/care to use it, it is almost meaningless. That is why, as you will see, there is quite a bit of modularity and data buffering even when many times it is not guaranteed to be used.

I like to think about the architecture as 3 main components: sensors, database, and manager. The manager sits between the sensors and database and initiates transactions of data between them. While there are other elements to the entire system, which will be discussed in a later section, they either extend or are a part of these main components of the EnergyAware system.

3.1 Sensors

On the sensor level, where you place the Wemo Switches is very important. First you want to ensure that every plug in the space is equipped with a sensor. These our outermost measurement points ensure that no power usage is unaccounted for and we have a direct way of measuring total energy consumption.

The next two metrics for deciding to place on a switch or not are power consumption level and actuation demand. Power consumption is critical because we want to make sure to place sensors on low power consumer but it is not necessary to have to do so on large power consumers. This is because it is easy to detect state changes in large devices just from looking at an outermost sensing point. This is shown in the drops and increases in Figure 2. The second metric is actuation demand. If you want to be able to remotely control some appliance then you want to make sure to put it on a sensor and, to be most effective, group together devices that you want to be able to turn off together, say the group of devices that you want to be off when you leave the house.

So, we categorize devices with Level 1 being the outermost socket sensors, Level 2 begin the devices we want to actuate and measure, and Level 3 being devices we just want to measure. This organization comes in handy when it comes to complex implementations.

The other categorization is devices that we want to be off when we are not at home. So we also label devices with a tag that identifies if they are occupancy dependent or not.

$$PlugHierarchy \begin{cases} Socket1 \begin{cases} Light1-d \\ Laptop \\ Monitor1-d \\ Monitor2-d \end{cases} \\ Socket2 \begin{cases} Phone \\ Light2 \end{cases} \end{cases}$$

We have here an example plug structure.

3.2 Database

Our databases are an essential part of the system architecture as the allow for data and information flow. Most of the information is passed as JSON objects as that is what works well with MongoDB, the database we chose for this implementation. We have a few databases that each serve a separate but essential purpose in the entire system's functionality.

The first two databases are the `wemo.names` and `wemo.ips` databases. The names database designates which switches on the network to be considered in our system. Because the Wemo switches are WiFi enabled, anyone on the network can see them. Many times campus residents share a Wifi network and our system would not want to include all the sensors on the network, because they might not all be owned by the user, so we have them specify which sensors to include. The second database asks for the IP addresses of all network connected devices which a user possesses. We will discuss the purpose of this later in the paper.

The next databases is `wemo.todo`. This is essentially our instruction stack. Instructions are popped off the stack and execute by the central system. This database is extremely restricted because it actually poses security and privacy concerns due to the ability of changing states of a users devices.

The final two databases are about information storage. These are `wemo.reading` and `wemo.state`. Our reading database is what we write all the power reading from each of the sensors to. Our state databases is where we write state change events to.

The databases feed other components on top of the system that look to it for information. While the manager also plays a part in this, both need to ensure that the data that is being passed backwards is presentable, whether that be by smoothing, analytics, or other tools.

3.3 Manager

The manager is the central piece to the entire system. It is what connects all the components and manages the registered devices. Since it is so central, we have made it as fault tolerant as possible. However in the case that it does break, we don't want to have the system down for long periods and without anyone knowing, so we have a fault alert system that will alert users before the script shuts down.

The manager runs on the Intel Edison boards and requires constant power. But fortunately, in the case of power outages or faultiness, the script will automatically start itself again and because of the way it is written, it will miss/lose little to no information at all.

4. CORE IMPLEMENTATION

Our implementation was primarily in Python and various REST interfaces. I chose Python for this project because of my experience with it and also for how easily it works well with many of the technologies (software and hardware) that we are working with.

4.1 Sensors

The Sensor setup is the one part that we weren't able to integrate into our system. The setup process is long and tedious and encounters many issues. We are forced to use the Wemo application for iOS and Android to setup the sensors with a name and Wifi connection.

However after that point, but before you power on the Intel Edison board, you must register the devices through the entirely web interface and proceed with the setup of your "personalized" system. At that point you will plug in the Intel Edison board and the script will import all the data you entered to initialize the system properly.

4.2 Databases

We have discussed how each of the other components uses and interacts with the databases, but have never discussed the format in which it holds data and the schemas that it adheres to. Our databases are listed below:

```
wemo.reading
wemo.state
wemo.todo
wemo.err
wemo.names
wemo.ips
```

The structure of the databases adheres to the following standards, but ignores what is not pertinent to the individual databases purpose:

```
reading: {"time":-,"name":-,"nick":-,"data":- }
state: {"time":-,"name":-,"change":on/off}
todo: {"data":-}
err: {"time":-,"id":-}
names: {"name":-,"nick":-,"level":- }
todo: {"ip":-}
```

4.2.1 *REST Interface*

At first we tried to build our system off the existing HTTP interface that exists for MongoDB. This worked well until we realized that it was limiting our JSON return results to at most 1000 rows. Given that we read our sensor data at a high granularity, this data base fills up past 1000 rows quite quickly.

Many times this does not become an issue, especially if we are interfacing with that database with Python. But when we are using Javascript, as we do in our front end, we need to have a way to serve all the data as a JSON. That is why we have a REST interface that runs on our DigitalOcean server, that serves the requested JSONs and preprocesses the data so it is readable and easier for the front end developer to work with.

4.3 Manager

The manager is our `main.py` that is constantly running on the Intel Edison board. Since this a main component of the system, it stresses efficiency and fault tolerance. To start the manager and therefore the EnergyAware system, all you must do is supply power to the system. It will automatically startup the codebase and send a message to verify it has started up properly since the Intel Edison has no GUI to display errors or issues.

4.3.1 *Initialization*

The first thing it does is gather the necessary data from the various databases. From this it constructs a dictionary that maps `Sensor Name` to `(Sensor Level, Nickname)`. This Sensor Level concept is discussed in the architecture section but essentially it represents its place in the sensor hierarchy. Many times the Sensor name is complex or random, so the Nickname is a more user friendly way to reference the sensor. The nickname can be something like "Laptop", "Light1", etc.

Now, since many time when taking input from the user we would like to be able to handle them referring to sensors by their nickname, we also have a dictionary `nicknames`. This simply maps a nickname to the sensor name which helps other later components later.

4.3.2 *ouimeaux*

The ouimeaux libraries allow Python to interface with the Wemo Insight sensors. The ouimeaux module is Open Source so we are allowed to use it in our system.

Using this system, the next step in our process is to do a scan on the network for *any* Wemo device. That is why it is essential, above, to keep track of all Wemo switches that the user registers as their own. So after the scan we only keep the registered switches as create our `switchmap` that maps switch names to the actual switch object. This switch object is the one we interact with for control and information on the switch.

With the ouimeaux library we request various pieces of information from the switches. In our implementation we query for 3 pieces of data: todaykwh, currentpwr,todayontime. These pieces of data give as an instantaneous sense of usage with power, a cumulative usage estimate with power, and also a sense of the "on time" of the devices we use.

4.3.3 *Data Accumulation Loop*

This loop will execute every x seconds. This x is given as a command line argument when starting the main script. In each iteration of the loop we do the following:

1. Get data from sensors
2. Detect sensor level state changes
    (a) Check if 0 to nonzero (on)
    (b) Check if nonzero to 0 (off)
3. Increment total if level 1 sensor
4. Write all data to database

We keep track of the total data (across all assigned Level 1 Sensors) because that is probably what a user would like to see. The system is designed to get users the energy information that is usually inaccessible. They are probably most interested in their overall usage patterns.

4.3.4 Occupancy sensing

While the occupancy is part of the loop pattern above, we give it another section here because of its significance to the system as a whole. Our occupancy sensing isn't as rigorous or thorough as other occupancy sensing projects. A user will have a registered set of devices and IPs and we monitor the presence of these devices on the network to "measure" occupancy. This is done by simply calling `subprocess.call(['ping', '-c', '3', ip])`.

A challenge of the occupancy sensing, which was only discovered recently, is that in iPhone's, the network hardware will go into sleep state when the phone goes to a sleep state. This was originally throwing off all our occupancy readings to be when the device is in use. However we now use a clever method (this was not discovered by me and the source is cited below). The `port 62078` on the iPhone is known as the `iphone-sync` port and is used to connect to iTunes and sync. We can issue a `tcping IP 62078` and will receive a response even when the phone is in a sleep state.

I am also working on incorporating bluetooth scanning for a set of devices to do the same. We came upon this idea by realizing that a later component, Physical Web sensors, is also dependent on bluetooth so it is likely that a user of this system will have their bluetooth on as well. The bluetooth sensing works if you run the system from your desktop but is not yet implemented on the Intel Edison. However, this bluetooth component is not essential to the performance of the EnergyAware system.

The occupancy sensing is not done at the same granularity as the sensor reading. This is under the assumption that the user of the system will not be entering and exiting within a matter of seconds and even if they did, accounting for it would have little effect on how our system functions. This design decision also is conscious of not having to send unnecessarily many pings to your devices.

4.3.5 Fault Tolerance

Given that we are running the manager on an Intel Edison or equivalent board, we are not giving feedback to the user. Therefore, to give them an update the the system has started, we send a text message alerting "System Start". We also send a message alerting "System Stop" if the script ever stops running. This way, they can assess and report an error to be resolved, and there is never a situation where the user believe the EnergyAware system is running when in fact it is not. Finally, as discussed in the first paragraph, we need to give alerts when we hit errors, however not just to the users, but to us as developers as well. So before exit, the script will also write to an error database with the MAC address of the machine its running on. This allows us as the developers to diagnose and fix issues.

Since many of the calls and data reads are dependent other networked hardware, there is a chance that something malfunctions and we can't get a reading. This is not in our codebase, rather in the Python libraries and actual Wemo devices, so we can't fix the issue but instead must learn to handle these errors. The first way we do this is by assigning timeouts to all network dependent functions. This ensures that we never get stuck waiting when a sensor has dropped off line and can continue with the next iteration of the loop. Also, given that we don't know the errors or steps that the ouimeaux library takes to read data from the sensors, we also have wrap all their function in `try...except...` handlers to prevent the script from crashing.

4.4 Instruction Stack

The manager as described above is limited to outward interaction. It gets data from the sensors, it sends data to databases, and it sends messages to the user. But at its current state, there is no way for anything outside the codebase to interact with the system. This why we have the Instruction Stack and why its an integral part of the interaction with EnergyAware

The instruction stack is implemented itself as a database. The manager loop will every so often (not at the same rate as sensor readings) will read and delete all the data from the database. Then according to some protocol it will execute these instructions. We will talk about the various functions of the instruction stack later in this section. Below are described the various methods of interaction, however keep in mind that this can be generalized to any method of interaction given how generally our instruction stack structure is designed. Being a set of string representing instructions, any device that can post to a database is capable of being programmed to interact with the Energy Aware system.

4.4.1 Twilio

One of the various methods of interaction in the EnergyAware system is through text messaging. To allow us to send and receive messages, we use the Twilio service. Twilio gives us a dedicated phone number that we can interact with using HTTP. On our Digital Ocean droplet (which is discussed in a later section), we constantly run a HTTP server using Python that takes the text received from the message, parses it properly, and inserts it into the Instruction Stack database.

The first text message based interaction is changing the state of devices. The instruction for this is of the form:

```
{turn,switch...} {name,nickname} {on,off}
```

The next message based interaction is getting the state of the devices. And to do that, we pass:

```
is {name,nickname} {on,off}
```

Now we want to make sure, from a user interaction/usability perspective, that we are selective in what we automatically send the user. We want to give the user alerts and messages, but we don't want to bombard them with unnecessary messages because this reduces the attractiveness of using EnergyAware. So we allow them to choose preferences for how they will receive messages.

We can also use the Twilio system for sending two other alerts about the data that we collect. As we will later discuss in the front end, we allow the user to set daily energy usage and time usage quotas. When users reach certain percentages in their quotas we can send them text message alerts that they have used that percentage.

Also, we can alert users when they are not at home that they have left devices on. We do this for devices that are marked

4.4.2 Security

While security is an important part of the entire EnergyAware system as a whole, it is particularly important here. Security of the instruction stack is essential, because without it, we could have tampering with devices on and off state or overloading the Instruction Stack. We have various methods of securing, as discussed below.

The first method is simply just with username and password on the database. All elements that need to access the database to read or write know the username and password and none of these elements are forward facing, meaning they are either embedded in a script or hidden by a `.htaccess` requirement.

Now the next layer of protection involves the `iptables` that restrict the IP addresses that the mongod server accepts connection from. This reduces the damage that an attacker trying to bombard the database could do.

Finally, a security measure that is natural implemented in Twilio, not done by us, is restricting the phone numbers that we can send messages to. But the measure that we have added is restricting the phone number that can message our Twilio number by filtering out messages that come from unregistered numbers.

4.4.3 Integration with Manger

The code for how this operates with the manager is below and is quite straightforward:

```
... Outer Loop ...
s=db.find()
s=[i["data"] for i in s]
for instruct in s:
        i=[ins.lower() for ins in instruct]
        execute(i)
db3.delete_many({})
```

5. FURTHER IMPLEMENTATION

So everything discussed previously consists of the barebone essentials of the EnergyAware system. But any useful and complete system is built to be expanded and extended upon. We kept this in my mind while designing the system as you will see in the following sections.

5.1 Updates

We have a system in place so that even if the EnergyAware is already deployed, we can remotely change the code when we receive reports of errors. Already described in an earlier section is the `wemo.err` database that we can monitor to see errors and on which deployments of the system they occurred on.

With that we can make changes where necessary. The main.py script will only execute after sending a request with the current version number to:

`http://www.nishadg.com:8082`

And if the version is not the current one, it will get the next version main.py at:

`http://www.nishadg.com/update`

There is a security flaw here that I am working on researching how to be able to serve an update without allowing a user to look at what it consists of.

5.2 API

The EnergyAware system was designed so that 3rd parties could write "applications" on top of the underlying system. The term applications does not exactly fit here, rather you would be able to add your own routine that will execute along side the existing codebase. We here provide a Documentation of the API for this programmable additional routine.

EnergyAware will pass you the following data structures as described earlier in this paper:

1. switchpair: sensor name to sensor object
2. nicknames
3. devicemap: sensor information

Now the EnergyAware system also requires some restrictions on the "applications":

1. integer exectime
2. function appname
    (a) Must take 3 parameters as described above
    (b) Must execute in exectime seconds
    (c) Must not change passed argument

All this must be packaged in a file named appname.py. And all the apps must be stored in directory (at the same level as the main.py) named `apps`. Sometimes you will need to include an empty `__init__.py` file in the app and upper level directory

Now in our main script, we will first import all these "apps" and store the corresponding functions and exectime's in a list. We have a user specified loop interval, but we want to ensure that the sum of the exectime's is not more than the interval, since this could lead to unexpected side effects. So, we sort all the functions by their exectimes and execute only up until the point where the sum of the exectime's is less than the user defined interval. This is all under the assumption that if a user wants to use more applications, they will choose a larger interval.

Now, to enforce the exectimes and ensure that an app doesn't just maliciously take up time, we use the same timeout library as we used on our own code and apply that to user code.

The protocol described is implemented as such:

```
f=os.listdir("apps")
f=[t for t in f if ".py" in t]
apps=[]
for app in f:
        name=app[:-3]
        modname="apps."+name
        t=__import__(modname,fromlist=[''])
        apps.append((t.appname,t.exectime))
```

5.3 Data Analysis

The ML portion of the project was an element that we were not able to complete to its entirety. We have worked out analyzing the data and learning from it, but we have not implemented feedback from it. So, the results of learning are

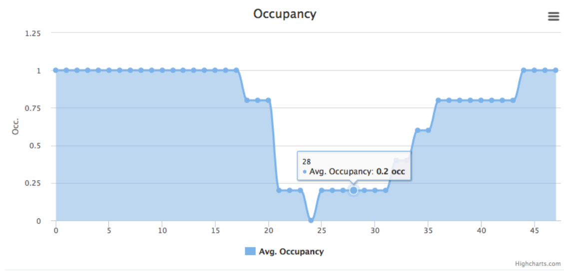

Figure 2: Occupancy Average

Figure 3: Physical Web Beacon

not fed backwards to the system so that it can adapt. So in this section we will discuss the analysis we did of the data.

The first thing we did is look just at occupancy data. With occupancy data you can look for routines. Our occupancy data is represented by a graph that takes on values of 0 and 1 over time. Alone this data is not interesting, but averaging all these together gives you a smoother average occupancy over time. An example this averaging over just a week of my occupancy data is shown in Figure 2.

5.4 Physical Web Beacons

The next element was inspired by what I will be working on this summer. The concepts of Smart Home, Smart Meter, etc. are extensions of what is commonly referred to as the Internet of Things. The Physical Web beacons are Bluetooth beacons that transmit URLs. The concept is quite simple, but that is for a good reason, because it is a concept that can generalize to everyday objects.

For our project we wanted to extend this concept to our outlets and power usage in our home. The inspiration for my entire project was let people have access to this new set of data that was previously entirely unaccessible. The Physical Web beacons are trying to create a new level of interaction, and in this application, we want to allow users to be able to "interact" with their home power supplies in a way that was previously not possible.

Currently the beacons are larger and not integrated into the power meter, as shown in Figure 3. But we hope that later, this kind of technology will be built into the sensors, as it is quite low power and efficient.

What the beacons are programmed to do is transmit a URL that represents its corresponding power meter. And this URL essentially represents a "homepage" of that power meter. It gives us readings of total power usage, a graph of its power usage, and the ability to switch a sensor on/off. We wanted to keep it simple since it was a user friendly "insight" into the outlet. A sample of what this energy dashboard looks like is shown in Figure 4.

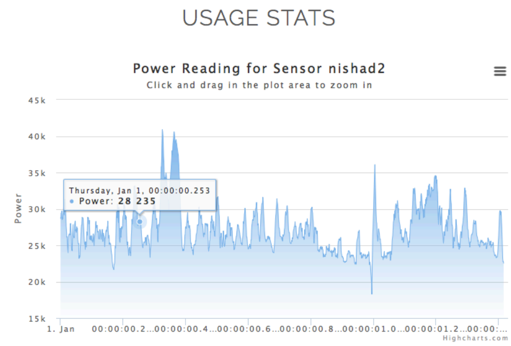

Figure 4: Sensor Dashboard

5.4.1 *Security (Extended)*

It was at this point that we realized that since we are transmitting a signal about the power meter that anyone could see, we needed to add another higher and complete level of security. So, we turned the entire system into an account based system where you needed to register and login before being able to access any data.

5.5 Front End

As just described above, at `www.nishadg.com/wemo`, you have must first login and be authenticated. You then have access to the Energy Dashboard.

You will first see a list of all the sensors that we have data on and short profiles about them. This is supposed to give you a way to compare these sensors, when compared to the individual view that the Physical Web beacons give you. They also have links to be able to see the individual, in-depth dashboards.

The next parts are to be able to see an overall view of your energy and power usage habits and characteristics. These include the total power usage readings graph. This gives you a sense of how your power usage varies over the day, over the month, etc. The Highcharts API has a useful graph mode that lets you zoom in and out to get a better view

In our midterm presentation, we made it a point that we wanted to try to change our users behavior. And, two methods we saw of doing this was by using Tradeoffs, Money, and Competition. Below we illustrate our various means of doing this.

In our abstract we discussed this concept of energy proportionality. If we graph the power consumption of a user with respect to occupancy (though occupancy in our sense is a discrete measurement), we can look at this line and have a sense of energy proportionality. While the slope of this line does't exactly tell us the energy proportionality, since the y-intercept plays a part, it does give a sense of how much changes when a user is home when compared to a user not being home. So our measure of energy proportionality is as follows:

$$AHU = \frac{(M_1 + M_2 ... + M_{10})}{10}$$
$$ANHU = \frac{(M_{11} + M_{12} ... + M_{20})}{10}$$
$$Score = (AHU - ANHU)/AHU$$

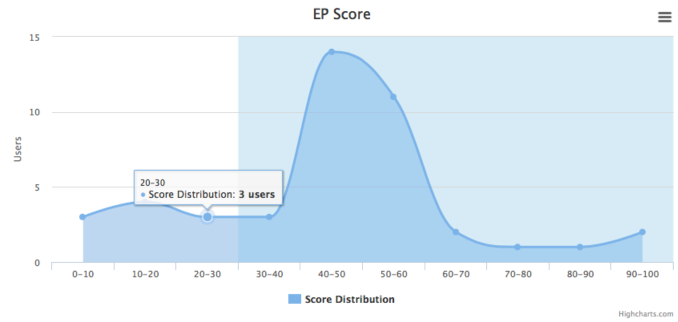

Figure 5: Energy Proportionality Score Distribution

Where AHU is the average home usage, ANHU is the average not home usage and your score correlates to your percent decrease when you're not at home. The M values are random samples of home and not home power readings. We thought to just average all that data, but realized that if this system runs for longer periods of time, that is quite an expensive operation and is not entirely necessary.

The other metric we present is cost. Since we can't be too fine grained about a cost measurement since we do not have statistics about it, we just place the traditional \$0.12 per kWh and compute that on total usage. We also do that computation daily to give a sense of how much you aggregate in costs of electricity per day.

Now, we can add the competitive element to this data by presenting the data to the user properly. In Figure 4, we have created some fake data for users since our system has not had many users. You can see the distribution of scores and the shaded area represents all the users that are doing better than yourself.

6. USER STUDY

We hope to deploy this system to many users. But obviously before we can do

7. CONCLUSIONS

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the LaTeX book, the citations in this paper are to articles which have nothing to do with the present subject and are used as examples only.

8. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

9. REFERENCES

[1] M. Bowman, S. K. Debray, and L. L. Peterson. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, November 1993.

[2] J. Braams. Babel, a multilingual style-option system for use with latex's standard document styles. *TUGboat*, 12(2):291–301, June 1991.

[3] M. Clark. Post congress tristesse. In *TeX90 Conference Proceedings*, pages 84–89. TeX Users Group, March 1991.

[4] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[5] L. Lamport. *LaTeX User's Guide and Document Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[6] S. Salas and E. Hille. *Calculus: One and Several Variable*. John Wiley and Sons, New York, 1978.

APPENDIX

A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure *within* an Appendix, start with **subsection** as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.2 The Body of the Paper

A.2.1 Type Changes and Special Characters

A.2.2 Math Equations

Inline (In-text) Equations.

Display Equations.

A.2.3 Citations

A.2.4 Tables

A.2.5 Figures

A.2.6 Theorem-like Constructs

A Caveat for the TeX Expert

A.3 Conclusions

A.4 Acknowledgments

A.5 Additional Authors

This section is inserted by LaTeX; you do not insert it. You just add the names and information in the `\additionalauthors` command at the start of the document.

A.6 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

B. MORE HELP FOR THE HARDY

The sig-alternate.cls file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of LaTeX, you may find reading it useful but please remember not to change it.