SystemSens: A Tool for Monitoring Usage in Smartphone Research Deployments

Hossein Falaki
CENS, UCLA

Ratul Mahajan
Microsoft Research

Deborah Estrin
CENS, UCLA

Abstract – By deploying several research applications, we found that capturing usage context (e.g., CPU and memory) is highly valuable for debugging and interpreting results, even if that context information appears unrelated to the application. We have developed a general tool called SystemSens to help researchers capture usage context in their deployments in an extendible way. This paper describes and motivates the design choices underlying our tool and evaluates its overheads in terms of phone resources and data.

Categories and Subject Descriptors

C.0 [**System architectures**]:

General Terms

Design

Keywords

Android, Smartphone, Deployment

1. INTRODUCTION

Using smartphones as a research platform is challenging [9]. Several engineering barriers such as closed and fragmented platforms [10] make it hard to develop software that works robustly on a range of devices. In addition, the diversity of usage patterns across users [6], makes it difficult to debug unexpected behavior, interpret results, and draw general conclusions.

Through deploying and supporting several research applications on smartphones, we find that capturing the broader usage context greatly simplifies some of the challenges. That is, research applications should not only capture information of direct interest (e.g., location for an application interested in user mobility) but also other information on how the smartphone is being used by the user (e.g., CPU, memory, battery, etc.) By doing so, researchers can better understand the environment their application operates in and interpret and qualify the results more accurately. For instance, if an application interested in location information also captures CPU and screen activity, it can better distinguish between users that are sedentary versus those that leave their smartphones on the desk for long periods.

To help research applications capture usage context we have developed a tool called *SystemSens*. It collects and logs smartphone usage parameters in the wild in an unobtrusive, and expandable way. SystemSens consists of an Android logging client and a visualization web service. This tool has been used in several deployments, where it has helped us better understand users' interactions with research applications [2, 8]. While the research supported by SystemSens has been published and so has an analysis of usage patterns [6], the focus of this paper is describing its design and our experience with it.[1]

During the past two years SystemSens evolved through three phases. Initially (v0) it was a simple battery and screen status logging tool that kept the traces on the SD card. The next major version of SystemSens (v1) recorded rich operating system and network information such as packet headers, and uploaded data in XML format to SensorBase [3]. This version used low-level Android libraries and could thus run only on developer phones, which proved to be a significant limitation because recruiting users became harder. In the next version of SystemSens (v2), we dropped some of the low-level logging capabilities to be able to run on stock Android phones. From this version we also started uploading data in JSON format to a collection and visualization server. The current version of SystemSens (v3) allows other (third-party) applications to log additional sensors through SystemSens, thus offering an extensible platform for monitoring smartphones.

SystemSens is designed to be unobtrusive—it has no user interface to minimize impact on usage, and it has a small footprint in terms of memory, CPU, and energy consumption. When having to choose between getting rich, low-level information and portability, we chose portability to run on any Android smartphone. Based on our experiments, we find that event-based data logging is more efficient than periodic polling. We also find that the primary energy cost of logging relates to how often the device is woken up, and the marginal cost of reading and storing additional sensory information is not significant. Thus, the designer of tools like SystemSens should optimize for maximizing sleeping and worry less about the amount of information being collected and logged.

MobiArch'11, June 28, 2011, Bethesda, Maryland, USA.

[1]The latest source code and API can be downloaded from *http://systemsens.cens.ucla.edu*.

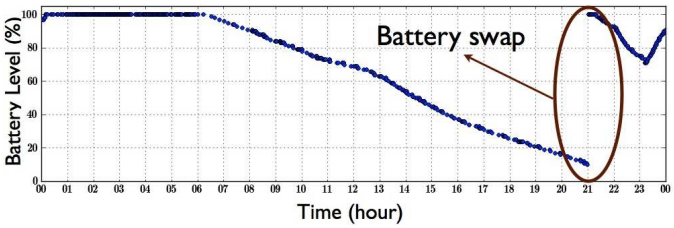

Figure 1: Snapshot of SystemSens battery graph of a user who used a backup battery.

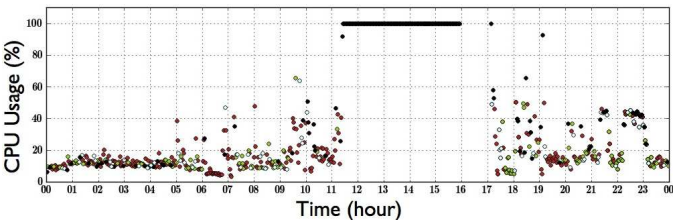

Figure 2: Snapshot of a SystemSens graph showing average CPU usage during one day for a user. Colors represent CPU frequency.

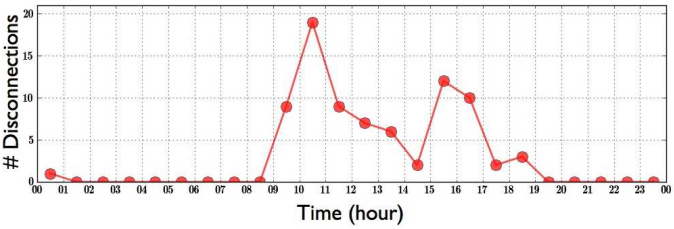

Figure 3: Snapshot of a SystemSens graph showing the number of cellular disconnection events per hour during one day for a user with poor connectivity at work.

2. THE IMPORTANCE OF USAGE CONTEXT

In this section, we motivate the need for capturing broader usage context when deploying research applications. During the past two years, SystemSens was used in several deployments. During the summers of 2009 and 2010 it was used by about 30 high school students who ran a range of participatory sensing applications [2]. Within the past year, three pilot deployments of the Andwellness project [8] had SystemSens running on the smartphones of their users. These were in addition to several internal studies. In all these cases we found that data from SystemSens provided valuable insight into the usage context, helped us optimize our applications to consume less energy, and pointed us to external factors that were affecting our studies, but we were not aware of. We present a few examples.

2.1 Unexpected User Behavior

When using smartphones as a research platform, it is convenient to assume that the research application is the only one on the subjects' smartphones, because it is the only application that researchers can control. This assumption works most of the time, but when it does not, it is difficult to identify the culprit.

For example, on a memory constrained device, seemingly irrelevant issues may cause problems for a research application. In Android, if a new application is launched when there is not enough free memory, the system automatically reclaims memory from other applications, even if that leads to stopping some of the background services. In one incident, we had a location tracking application that was supposed to continuously reside in memory and run. When testing it in the lab, it presented the right behavior, and worked as expected for most users. We experienced problems with a few of our users, which we could not explain initially. But after looking at memory and interaction traces from their SystemSens logs, we found that those were active users with high memory usage. In another deployment, some of our users reported repeated phone reboots. Inspection of memory usage information from SystemSens showed that these were heavy phone users, who continuously invoked different applications, and therefore Android was under memory pressure on those phones, and that version of Android on that hardware platform resorts to rebooting the phone when memory is scarce.

In every deployment of research software we have encountered a few users with other forms of unexpected behavior that impacts the research applications. As an example, consider the graph in Figure 1. It is a snapshot of the SystemSens battery level graph over 24 hours for a user who used two batteries. The sudden jump to 100% at about 9pm, indicates switching batteries. We never considered this behavior when developing and testing our log upload mechanism that kicks in only when the phone is being charged. If battery swap behavior were dominant, no data would get uploaded (and we wouldn't be able to distinguish between the application not working and upload failures). Fortunately, this user sometimes also charged her phone, which led only to delayed uploads and not no uploads.

2.2 Debugging Battery Consumption

Using SystemSens in our deployments helped us better understand the impact of our research applications on the battery experience of our users. To do this, just monitoring battery information is not enough. Very often when trying to explain short battery life of the users we identified problems in our software and fixed it, but in many cases other information from SystemSens helped us identify external factors that were contributing to high battery drain.

For a few users, when inspecting SystemSens graphs we encountered unusual CPU activity as shown in Figure 2. We traced the problem to a bug in the GPS driver of that phone model. Under specific conditions, the GPS driver consumed 100% of CPU time at the highest frequency until no juice was left in the battery and the phone died.

For some other users we found an unusually high number of network disconnection events. Figure 3 is a snapshot of the SystemSens graph for the number of cellular disconnections for one such user who worked in a building with very poor wireless reception. SystemSens reported repeated disconnection events during day hours. When this happened the cellular interface consumed significantly more power, which resulted in much shorter battery life time.

In some other cases, we discovered that some of our users had ruptured batteries. Their SystemSens battery graphs showed that the batteries could not hold charge more than two hours — an unusually short life for that phone model.

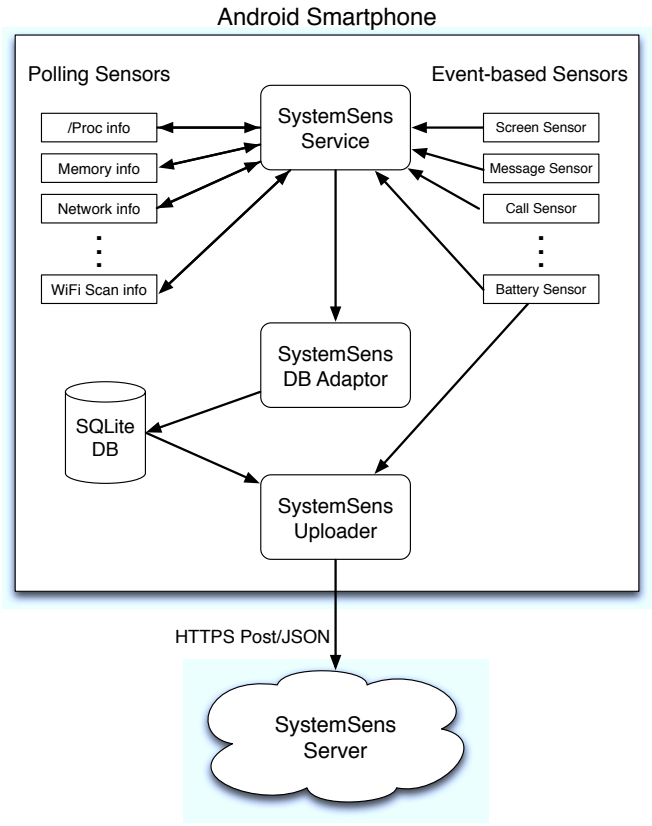

Figure 4: The architecture of the SystemSens client application.

When we inspected the phones we found the batteries had been ruptured.

In each of these instances, without access to the information from SystemSens, we could not have identified the real culprits. That would have led to unsatisfied users who would have blamed our application for the drain and likely uninstalled it.

3. ARCHITECTURE & DESIGN

In this section we introduce the architecture of SystemSens. We highlight our design decisions and the reasons behind them.

The principal goal that derives most of our design and implementation decisions of SystemSens is to keep a low profile in terms of resource consumption on the smartphone. Therefore we have designed SystemSens to minimize the amount of work on the client and delegate complexity to the server and (offline) analysis. We will refer to this choice as the *thin client principle*.

A second goal behind many design choices of SystemSens is to keep the userbase as broad as possible. Most importantly we abandoned some of the features of version 1 to be able to to run SystemSens on stock Android smartphones.

3.1 SystemSens Client

Figure 4 shows the architecture of the SystemSens client application. The SystemSens client continuously runs as a background Android service. We chose not to implement any user interfaces to minimize impact on usage. The client records and uploads a range of operating system events. Each group of related OS information is recorded by a virtual "sensor." SystemSens supports two types of sensors. *Event-based* sensors generate a log record whenever the corresponding state changes. For example, the screen sensor records the state of the screen every time it turns on or off. *Polling* sensors record the corresponding information at regular intervals. For example, every two minutes the average CPU and memory usage are recorded. Table 1 is the list of all the sensors currently supported by SystemSens. To minimize energy consumption of SystemSens we prefer an event-based sensor to a polling sensor to record the same information.

The main SystemSens thread is responsible for recording both event-based and polling sensors. All polling sensors are queried at fixed intervals (currently set to two minutes — we found this value good enough). In previous versions we polled the sensors more aggressively when the user was interacting with the phone. We found that variable intervals make visualization and analysis of traces more difficult. To improve the performance of the recording operation, all generated records are kept in a memory buffer that is flushed to an SQLite database table on the phone regularly in a separate thread.

The most energy and resource consuming task of the SystemSens client is uploading the records to the server because of high energy consumption associated with network transmission. Since none of our research studies require real-time data analysis, we designed the upload mechanism to upload only when the phone is being charged. Most smartphone users charge their phones overnight, which offers SystemSens enough opportunity to upload all the data. In addition, with this policy upload happens at a time when users tend to have zero interaction with their phones. This scheme would fail if a user charges her phone while it is turned off, or at a location with no network connectivity, or for users with multiple batteries who charge them outside the phone.

When the upload thread starts, it reads the local database in batches of 200 records, URL encodes them and posts them to the SystemSens server over HTTPS. If the post request succeeds, the records are deleted from the database with a single range query. We decided not to authenticate clients when uploading to be able to support users who are not registered on the server and keep the potential user base broad.

If for any reason, such as network disconnection, the server does not confirm receiving a batch of data, they will not be deleted. This approach further simplifies the logic of the client, but may result in duplicate records in the database. However, filtering duplicate records during analysis is trivial.

3.2 Data format

We chose the JSON format [1] for SystemSens data both on the server and the client. In both client and server databases JSON objects are stored as strings. This gives us the flexibility to add new types of sensors and data record types without the need to change database schema. In addition, handling JSON objects on Android is slightly more efficient than XML [12].

Name	Type	Information	Name	Type	Information
battery	Event	Battery level, voltage, temperature, health and charging status	servicelog	Polling	Start and end time of background services
call	Event	Voice call state	activitylog	Polling	Start and end time of applications
cpu	Polling	Contents of /proc/cpuinfo	callstate	Event	State of dialer application
celllocation	Event	ID of the connected cell tower	meminfo	Polling	Contents of /proc/meminfo
dataconnection	Event	State of connection to data networks	memory	Polling	Android reported memory information
servicestate	Event	Operator information	netlog	Polling	Contents of /proc/net/dev
network	Polling	Traffic statistics per interface	gpsstate	Event	State of Android GPS provider
wifiscan	Polling	Signal strength of visible WiFi APs	message	Event	State of pending unread text messages
callforwarding	Event	State of call forwarding	screen	Event	State of the screen
appresource	Polling	Memory and CPU usage of running apps	netlocation	Event	Course (network based) location
systemsens	Event	Start time of the SystemSens application	netiflog	Polling	Traffic statistics per applications

Table 1: List of default SystemSens virtual sensors

```
{'date':       '2011-2-22 0:42:56',
 'time_stamp': 1298364176416,
 'type':       'screen',
 'user':       '355060041008892',
 'ver':        '3.2',
 'data':       {'status':'off'}}
```

Figure 5: Example of a SystemSens data record

Each record contains the IMEI[2] or ESN[3] of the phone to identify the user, the timestamp in milliseconds in UTC, the human readable date in phone local time zone, and the version number of SystemSens. Each record contains a record type field. The type value is used to parse the contents of the data field which is itself a JSON object. Figure 5 is an example of the JSON object generated by the screen sensor.

3.3 SystemSens Server

Our server configuration consists of a MySQL database on a Linux machine running Apache. All server functionality is implemented in Python.

When the server receives a post request from a client it parses the JSON object and inserts the type, user, and date records along with the original JSON string in a database table. This table is indexed on time, user and date fields to facilitate fast search queries on the records.

To preserve privacy all server functionalities that read the data require authentication. Each login is paired with an IMEI or ESN, therefore each user can only access her data. The data is presented as a number of time graphs such as those in Figures 1 and 3. Each graph has time as the X axis and the user can control the range of the X axis using time controls. Graphs can be generated for each data type or combinations. Adding a new graph to the visualization service is as simple as adding a new function.

3.4 External Sensors

Other applications can act as virtual sensors for SystemSens, and log information through SystemSens by implementing a simple AIDL interface. SystemSens will treat the application as another virtual sensor and poll its values.

[2]International Mobile Equipment Identity

[3]Electronic Serial Number — equivalent to IMEI for CDMA phones.

Most of our research applications use this feature to closely monitor their resource consumption. Other research applications can use the SystemSens framework to upload and monitor their data. For example, a sleep survey application that uses accelerometer to detect when the user wakes up and asks questions regarding quality of sleep logs its accelerometer usage through SystemSens.

4. EVALUATION

In this section we present basic evaluation of the performance of the SystemSens client. We summarize statistics regarding the amount of data that SystemSens generates. Next we evaluate the impact of running SystemSens on the battery life of Android phones. CPU and memory usage of SystemSens is insignificant. On a Samsung Galaxy S smartphone SystemSens on average consumes about 3% and 2.5% of CPU time in user and kernel mode, respectively. It occupies about 4% and 3% of memory pages in private and shared mode, respectively.

4.1 Data Size

The amount of data that SystemSens generates for each user varies widely depending on usage and version of Android. When there is more interaction, more event based records are generated and on older versions of Android some of the virtual sensors are not accessible. Figure 6 is the CDF of the number of records per hour for two example users both with high-end phones running Android 2.2. The median for the first user is 408 and for the second user is 445.

The length of records primarily depends on their type. Table 2 contains the median length of records of each type in the SystemSens database. The median length of all records is 159 characters and the mean is 362.

Based on these statistics and further inspection of the existing records in the current database (more than 10 million records), we find that SystemSens on average generates about 2.5 MB of data for each user per day. On the high end this value is 5.5 MB and on the low end 0.75 MB.

4.2 Energy Consumption

To evaluate the energy consumption of SystemSens we report the results of two types of experiments. First, we directly measure power consumption of a phone when running SystemSens. Second, we measure the reduction in battery

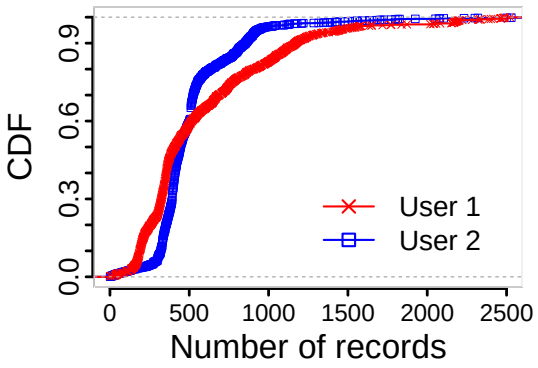

Figure 6: CDF of the number of records generated per hour for two example users.

Type	Length	Type	Length
activitylog	133	battery	224
callforwarding	153	call	145
celllocation	159	callstate	150
dataconnection	181	cpu	324
gpsstate	192	meminfo	459
message	146	memory	128
netiflog	314	netlog	2871
servicelog	449	screen	143
servicestate	235	network	149
appresource	4132	netlocation	223
systemsens	190	wifiscan	154

Table 2: Median length of different record types of SystemSens.

life time of a phone when running our tool. Both of these tests are common in the mobile computing community.

We placed a high frequency digital voltmeter in parallel and a current meter in serial with the battery of a new Samsung Galaxy S Android smartphone. We measured the current and voltage across the battery with a frequency of 50 Hz. For each experiment we measured power consumption for 10 minutes and report the mean power consumption in Figure 7. No other third-party application was running on the phone during these tests. We start measuring power a few minutes after the screen turns off.

The average power consumption of the phone when no background application is running is about 24 mW. When SystemSens is running the phone consumes about 43 mW. To put these numbers in perspective note that this device consumes about 500 mW when the screen is on, and about 1500 mW when it rings to an incoming call.

When there is no interaction with the phone, SystemSens consumes about 19 mW just for recording the polling sensors. Each polling cycle consists of waking the phone up, reading three files from /proc, querying some Android libraries for other polling sensors and finally writing the results, along with event-based sensor data received during the past polling interval, in the local database. To measure the power consumption associated with each of these steps we performed three additional experiments. The results are summarized in Figure 7. Reading three files from /proc consumes about 4 mW. Reading the other polling sensors consumes an additional 3 mW, and writing the data into the database consumes about 2 mW on average. These results

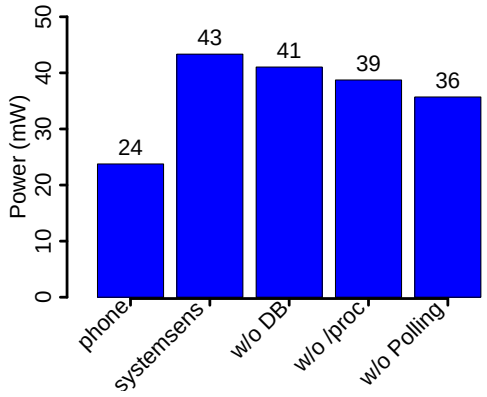

Figure 7: Power consumption of a Galaxy S smartphone with different versions of SystemSens.

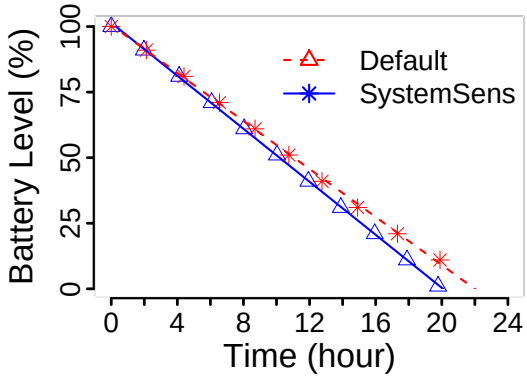

Figure 8: Lines fitted to battery level readings show the impact of running SystemSens on battery life of an old Nexus One phone

suggest that when the phone is woken up the marginal cost of polling additional sensors is insignificant. In addition, writing data into the persistent storage is not expensive in terms of power. Therefore, the most effective way of reducing energy consumption of SystemSens is increasing the polling interval.

Calculating battery lifetime using the declared battery capacity and measured power is inaccurate because it cannot account for many external factors such as variations in battery voltage [11] and user interactions. To get a realistic estimation of impact on battery life we performed our second experiment. We implemented a simple application that continuously records battery levels. We installed this application on a Nexus One with a full battery. We left the phone on a shelf until its battery died. We repeated this experiment with SystemSens and placed the phone in the exact same location. Figure 8 shows the recorded battery levels and the least squared fitted lines for both cases. It shows that SystemSens reduced the battery lifetime of this phone by about two hours when there is no user interaction. Note that the impact is different on different hardware platforms. We chose the used Nexus One rather than the new Galaxy S, because its shorter battery life made our experiments shorter.

Both these experiments capture worst case scenarios, because when a phone is actively used some of the polling events occur when the phone is already powered up by us-

age. By investigating SystemSens logs from many users, we find that when the phone is not being charged, between 8% to 20% of polling events happen when the screen is on or within 30 seconds (sleep timeout) after the screen is turned off. We found the distribution of this ratio robust to the exact choice of sleep timeout value. Furthermore, based on anecdotal evidence from our deployments, the impact of SystemSens on battery life of actively used smartphones does not disturb normal day to day usage of the phone.

5. RELATED WORK

MyExperience [7] is one of the earliest tools built to measure device usage and context information *in situ*. It runs on Windows Mobile smartphones and supports active context-triggered experience sampling. SystemSens is designed and implemented for Android smartphones. However, a Windows Mobile port of an early version of SystemSens exists [5]. Unlike MyExperience, SystemSens is a passive logging tool — we chose not to engage with users to minimize impact on usage. SystemSens users are able to tag interesting phenomena regarding their experience on the web interface.

LiveLab [4] is a similar research tool implemented for the iPhone platform. It measures usage and different aspects of wireless network performance. A key feature of LiveLab is "in-field programmability." The ability to update a logging tool in the field is critical in any real deployment. We realized this need and implemented a separate tool named *CENS Updater* that can update not only SystemSens but also all other CENS applications in the field. LiveLab is built to run on "jailbroken" iPhones and is capable of collecting a wide range of OS and network related information. We decided to limit the sensing capabilities of SystemSens, but to keep the potential user base as wide as possible by implementing it to run stock Android smartphones.

Both MyExperience and LiveLab focus on data collection on the phone. SystemSens is an end-to-end system that includes a web-based visualization and authentication service to provide feedback to users while preserving their privacy. In addition, the web interface greatly facilitate browsing and interpreting the data for researchers.

6. CONCLUSION

We have been developing and using SystemSens as a tool to capture usage and context related parameters in our research for the past two years. SystemSens has become a robust logging tool with a portable visualization server implementation. It is now an integral part of our research deployments. We released the source code [4] for other interested researchers.

In future we will continue improving SystemSens. Specifically we will address the potential upload problems by including checks to trigger opportunistic uploads when the default scheme fails repeatedly. We will also include mechanisms for third-party applications to log information in an event driven manner.

7. ACKNOWLEDGEMENT

We are grateful to the members of the Center For Embedded Networked Sensing that helped us with several deployments of SystemSens: Nithya Ramanathan, John Jenkins, Brent Longstaff, Kannan Parameswaran, Betta Dawson, Mohamad Monibi, and Joshua Selsky. Prof. Ramesh Govindan provided valuable advice to this project. Zainul Charbiwala helped us with energy measurements. We thank them all.

[4] Available at *http://systemsens.cens.ucla.edu/*

8. REFERENCES

[1] Javascript object notation. *http://www.json.org/*.

[2] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M.B. Srivastava. Participatory sensing. In *World Sensor Web Workshop*, 2006.

[3] K. Chang, N. Yau, M. Hansen, and D. Estrin. SensorBase.org – A Centralized Repository to Slog Sensor Network Data. In *Technical Report*. Center for Embedded Network Sensing, 2006.

[4] AR Clayton Shepard, C. Tossell, L. Zhong, and P.K. LiveLab. Measuring Wireless Networks and Smartphone Users in the Field. *HotMetrics*, 2010.

[5] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A first look at traffic on smartphones. In *IMC*, 2010.

[6] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. Diversity in smartphone usage. In *MobiSys*, 2010.

[7] J. Froehlich, M.Y. Chen, S. Consolvo, B. Harrison, and J.A. Landay. MyExperience: a system for in situ tracing and capturing of user feedback on mobile phones. In *MobiSys*, 2007.

[8] J. Hicks, N. Ramanathan, D. Kim, M. Monibi, J. Selsky, M. Hansen, and D. Estrin. Andwellness: An open mobile system for activity and experience sampling. In *Wireless Health 2010*, 2010.

[9] S. Keshav, Y. Chawathe, M. Chen, Y. Zhang, and A. Wolman. Cell phones as a research platform. In *MobiSys Panel*, 2007.

[10] E. Oliver. A survey of platforms for mobile networks research. *ACM MCCR*, 12(4), 2009.

[11] R. Rao, S. Vrudhula, and D.N. Rakhmatov. Battery modeling for energy aware system design. *IEEE Computer*, 36(12), 2003.

[12] J. Sharkey. Coding for life – battery life, that is. *Google IO Developer Conference*, 2009.