ShareSens: An Approach to Optimizing Energy Consumption of Continuous Mobile Sensing Workloads

Ahmed Abdel Moamen
Department of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
moamen@agents.usask.ca

Nadeem Jamali
Department of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
jamali@cs.usask.ca

***Abstract*—Smartphones and a growing number of wearable devices are equipped with a variety of powerful sensors. This has led to increased interest in developing of applications across a wide variety of domains including health-care, entertainment, environmental monitoring and transportation, which use sensor feeds to offer services. However, most of these applications require continuous sensing, which places a heavy demand on the device's typically limited battery power. This problem is further amplified as multiple applications attempt to monitor multiple sensors simultaneously.**

In this paper, we present ShareSens, our approach to opportunistically merge independent sensing requirements of applications. We achieve this using sensing schedulers for sensors, which determine the lowest sensing rate which would satisfy all requests, and then use custom filters to send out only the needed data to each application. Sensing requests made through the ShareSens API (which we have implemented for Android) are forwarded to the relevant schedulers which determine the optimum sensing rates to satisfy all requests. The paper presents the design and implementation of ShareSens, as well as results from our experimental work on the power savings that can be achieved by using it.

I. Introduction

Smartphones and the growing number of wearables have a variety of different sensors built into them. These sensors can be divided into three main categories: motion sensors (e.g., accelerometer, gravity, gyroscope, GPS, and rotation vector), position sensors (e.g., orientation, geomagnetic field, and proximity), and environment sensors (e.g., light, pressure, humidity and temperature). Sensors have different configuration requirements, which may include various parameters or settings that need to be specified, such as sampling rate, trigger conditions, identifiers, and calibration.

Sensors place different levels of power demand on a device. For example, GPS and orientation sensors are significantly more power-expensive than accelerometers. However, a continuous sensing workload – which requires sensors to be sampled for a period of time – can quickly drain the battery of a mobile device [1]. Such workloads are required for a variety of independent applications, such as those focused on personal health and fitness, but their impact on power consumption would be most pronounced in the context of contributing data to a significant number of crowd-sourced services [2]. One widely used method to reduce the energy consumed by continuous sensing is to decrease the sampling rate, the number of raw samples the sensors generate per unit time [3]. Although reducing the sampling rate can extend battery life, it can also lower sampling accuracy to a level that is less than ideal. We aim to (opportunistically) optimize the power consumption of sensors while meeting user-specified accuracy requirements.

Consider a scenario where two continuous sensing apps are running simultaneously on the same mobile device, requiring periodic samples from the a sensor, but at different sampling rates. The first is a body activity detection app, which needs accelerometer samples at the rate of $10Hz$, while the other is a background service for location-detection, which needs accelerometer samples at the rate of $20Hz$. Even though the $20Hz$ stream obviously *contains* a $10Hz$, existing sensor APIs on mobile platforms would set up two separate sampling requests, amounting to a cumulative sampling of the sensor at $30Hz$.

Our approach is to exploit the opportunity created by the overlap in requirements of sampling requests to optimize sensor use of power. We introduce ShareSens, an Android API for accessing sensor feeds, which enables sharing of overlapping feeds between applications. An optimizing sampling scheduler schedules the sampling of sensors based on the sensing requirements received from apps running concurrently. The scheduler opportunistically optimizes the effective sampling rate of each sensor, exploiting opportunities for different apps to share sensor samples when possible.

The rest of paper is organized as follows: Section II presents the related work. Section III and IV present the design and implementation of ShareSens respectively. Section V presents our experimental results showing power efficiency of the approach. Finally, Section VI concludes the paper.

II. Related Work

In this paper, we are interested in the effect of *continuous mobile sensing* on the power consumption of smartphones and wearables. We first present some representative continuous mobile sensing applications, and then discuss some existing platforms for serving such applications.

A. Continuous Mobile Sensing Applications

Many modern smartphones and wearables are equipped with a number of sensors, creating opportunities for application which can use feeds from these sensors. Of particular interest to us is continuous mobile sensing, which involves continual sampling of sensors over a period of time. A number of applications rely on continuous mobile sensing across a variety of domains, from healthcare (e.g., [4]) to social networks (e.g., [5]) to environmental monitoring (e.g., [6]) to transportation (e.g., [7]–[9]) and human activity recognition (e.g., [10], [11]).

Human activity recognition has opened the door for new types of applications in the field of personal healthcare such as fitness monitoring, elder-care support and chronic care [11]. Activity recognition systems typically have a sensing module that continuously collects relevant information about the body activities (e.g., walking, sitting, laying, etc.).

Recently, activity recognition has became a key component in several commercial products. For example, game consoles such as the Nintendo Wii[1] and the Microsoft Kinect[2] rely on the recognition of gestures or body movements to enhance the game experience. While these systems are originally developed for the entertainment sector, they have also been used for other types of applications such as personal fitness training [12]. Furthermore, some sports products such as the NikePlus[3] running shoes which integrate motion sensors and offer athletes feedback on user's performance.

Waze [8] is one of the largest community-orientated mobile travel applications with users volunteering information about their driving experience in real time, by reporting on congestions, delays, and gasoline prices. These reports then become the basis for information displayed on other drivers' maps (on their mobile devices), to help them make routing decisions. Similarly, TrafficPulse [7] combines sensor data from mobile devices with real-time traveler reports from frequent travelers, and then offers this information to other drivers in an aggregated form.

Mobile sensing has also been used by Uga et al. [13] in an earthquake warning system, which uses data from accelerometers present in many modern mobile devices to detect seismic vibrations. Devices send reports of likely seismic activity to a server which then aggregates the reports received to send out warnings.

[1] http://www.nintendo.com/wiiu

[2] http://www.microsoft.com/en-us/kinectforwindows/

[3] https://secure-nikeplus.nike.com/plus/

B. Mobile Sensing Platforms

Energy consumption in mobile phones is a well studied research topic in literature from various perspectives [3], [14]–[16], each offering a different approach. To our knowledge, there is no existing approach to economize by enabling sharing of sensing data between different applications.

Early efforts to build frameworks for mobile sensing applications have tended to trade off accuracy for lower power usage by implementing algorithms which reduce the amount of sensor data. Other frameworks [17], [18] have tried to reduce power usage by opportunistically offloading sensor data processing to back-end servers in the cloud infrastructure. Other techniques rely on adopting a variety of duty cycling techniques that manage the sleep cycle of sensors on the device in order to trade off the amount of battery consumed against sensing accuracy and latency [19]. The drawback of these approaches is that they are not applicable to continuous sensing scenarios.

Adapting sampling rates for human activity recognition is a well-studied research topic. SpeakerSense [20] uses a low sampling rate to detect whether a speaker exists. It switches to a high sampling rate when a speaker is detected. Similarly, SociableSense [21] lets the sensors operate at a high sampling rate only when interesting events happen. If there are no interesting events, the sensors are set to operate at a low sampling rate. Although both SpeakerSense and SociableSense use different sensor sampling rates for different contexts, our approach is different as we quantify the optimal sampling rate based on the current sensing requests while considering user accuracy requirements.

Some projects, instead of adapting the sampling rate, achieve energy savings through optimizing sensor duty cycles (i.e., periodic sensing and sleeping instead of continuous sensing). EEMSS [22] is an energy-aware framework for human activity recognition. EEMSS tries to achieve energy savings by shutting down unnecessary sensors, and carefully selecting sensor duty cycles. A sensor management scheme is used to determine the minimum sampling lengths and intervals for a set of sensors to recognize user states and to detect state transitions. Mercury [23] uses similar strategy to reduce energy consumption by disabling and enabling sensors dynamically. ShareSens addresses the orthogonal concern of sharing sensor feeds.

With the growing popularity of continuous sensing, it has also attracted the attention of hardware designers to

develop hardware processors to support it. For example, the Little Rock project [24] at Microsoft Research is developing energy efficient co-processors for mobile device, dedicated to the task of continuous sensing. The tasks of duty cycle management, sensor sampling, and signal processing are offloaded to the new co-processor, allowing the primary CPU to sleep more frequently, saving the overall power consumption. The Little Rock architecture gives programmers the flexibility to choose between the primary processor and the co-processor for the various tasks in their applications, but it also makes application development more challenging.

III. ShareSens Design

We have designed ShareSens using the Actor model [25] which is an increasingly influential model for concurrent systems. Actors are autonomous concurrently executing primitive agents (i.e., active objects) which communicate using asynchronous messages.

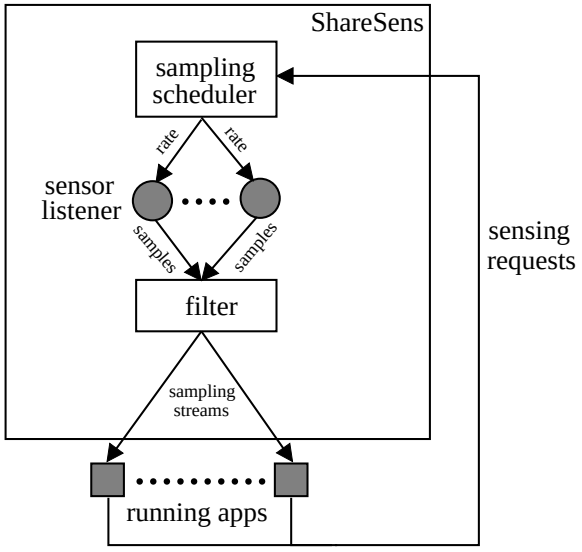


Figure 1. ShareSens Architecture

Our design of the ShareSens is shown in Figure 1. An optimizing *sampling scheduler* is used to schedule the sampling of each sensor based on the sensing requirements received from apps being served at the time. The scheduler attempts to optimize the sampling rate of each sensor exploiting opportunities for different apps to share sensor samples when possible.

When the scheduler receives a new sampling request, it checks if the current sampling rate – sufficient for serving all currently served requests – can also satisfy the new sampling rate being requested; if so, it uses the existing sampling stream; otherwise, it changes the sampling rate to be high enough to accommodate the new request. The new sampling rate can be computed by finding the greatest common divisor of the existing and the newly requested sampling rates.[4]

The sensor listener is responsible for sampling sensor data according to the sampling rate received from the sampling scheduler. However, because sensor samples are for all apps, there is a *filter* to extract the required samples to be sent to the different apps.

Algorithm 1 Sampling Rate Adaptation Algorithm Used in ShareSens

```
1:  procedure SENSOR SCHEDULING
2:  Input: sensor name (s) and sampling rate (r)
3:  Output: sensor data stream
4:  /* check if s is already scheduled */
5:  if SamplingScheduler.isSensorFound (s, r) is false then
6:      SamplingScheduler.add(s, r);
7:      create a new sensor listener actor for s
8:  else /* if s is already scheduled */
9:  /*find the GCD between r and current sampling rate*/
10:     newRate = GCD(SamplingScheduler.currentRate, r);
11: /* adapt the sampling rate */
12:             SamplingScheduler.adaptSamplingRate(s, newRate);
13: end if
14: filter sensor data
15: send sampling streams to apps when the sensor listener detects an event
```

Algorithm 1 shows the steps followed by the scheduler to find the optimal sampling rate for sensing requests being served at the time. Each sensing request specifies the sensor to be sampled, as well as the rate at which it should be sampled. When a new request is received, the scheduler checks if the sensor is already scheduled; if so, it merges the current sampling rate with the GCD of the inverse of the current sampling rate and the new rate; otherwise, it sets up a new sensor listener to the requested sensor.

IV. Implementation

A prototype of the system described in the previous section has been implemented as an Android app. As Figure 2 shows, our implementation is built over Actor Architecture (AA) [26], a Java library and runtime system for programming and executing distributed actor systems.[5] We have ported AA to Android OS for

[4]Appropriate granularity recommendations – such as the Android API's preferred sampling rates for typical use of $5Hz$, $16Hz$, $50Hz$, $100Hz$ – can be placed to decrease the likelihood of very high sampling rates.

[5]More precisely, it is an extension of AA for supporting resource-bounded actor computations based on the CyberOrgs model [27].

supporting the mobile app.

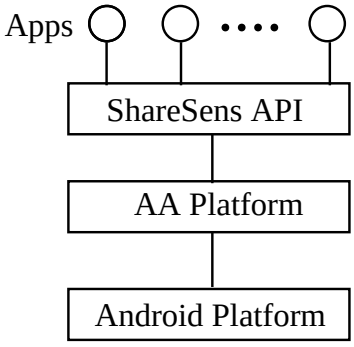


Figure 2. ShareSens Platform and APIs

A. Sensor APIs in Android

Android has a sensor API to read sensor data, which provides access to event handlers to capture sensor events. Android provides two event handlers through an interface called `SensorEventListener`: an `onSensorChanged` event is triggered when a sensor reports a new sensor value, and an `onAccuracyChanged` event is triggered when a sensor's accuracy changes.

The `SensorManager` class implements an Android service which provides various methods for accessing and listing sensors, and registering and unregistering sensor event listeners. To start up the sensor service, an app has to retrieve a handle to `SensorManager`, and uses the handle to make registration or unregistration calls for its event handlers.

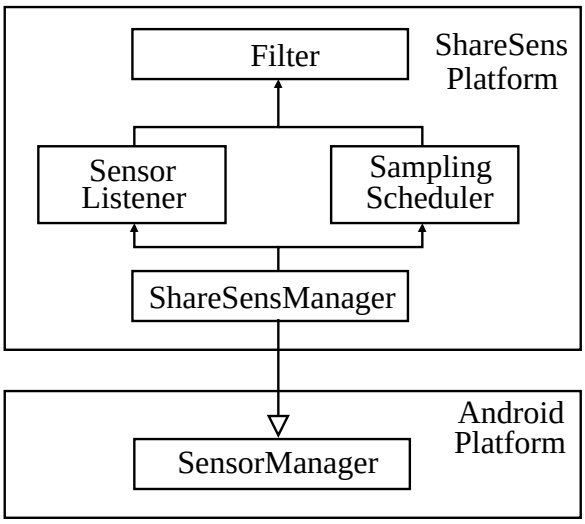


Figure 3. ShareSens Class Diagram

B. ShareSens API

As shown in Figure 3, the `ShareSensManager` class – which extends the Android platform's `SensorManager` abstract class – implements a background service to host the actors running on the system, and defines methods for accessing and managing sensors. At runtime, the `ShareSensManager` is responsible for computing the sampling frequency which would satisfy all requests, and keeps the `SensorServiceActor` informed of it. A `SensorServiceActor` encapsulates each sensor, and collects samples. Once an app registers a `SensorListener` actor for collecting sensor data, it begins receiving a custom stream of sensor data collected by the `SensorServiceActor` (which encapsulates a sensor). The customization is done by a special filter actor which knows the requirements of all sampling requests, and accordingly extracts them from the feed received from a `SensorServiceActor`, and forwards them to the relevant `SensorListeners`. A `SensorListener` passes the the feed it receives from the filter to its app. Both control and data communication happens through asynchronous message passing.

As discussed before in the design section, `ShareSensManager` restricts the sampling rate to be adjusted between a set of discrete values: (i) SENSOR_DELAY_NORMAL ($10Hz$) which is suitable for screen orientation changes; (ii) SENSOR_DELAY_UI ($20Hz$) which is suitable for the user interface; (iii) SENSOR_DELAY_GAME ($40Hz$) which is suitable for games; and (iv) SENSOR_DELAY_FASTEST ($80Hz$) which is the fastest rate at which sensor data is provided.

Figure 4 shows a code snippet that implements `ShareSensListener` and uses `ShareSensManager` to register an accelerometer sensor with the normal sampling rate (`SENSOR_DELAY_NORMAL`). A reference to the sensor server is obtained in order to identify the sensors available on the device. This requires creation of an instance of the `ShareSensManager` class by calling the `getSystemService()` method and passing it the `SENSOR_SERVICE` argument. A reference to the accelerometer sensor can be obtained using the getDefaultSensor() method by passing in the type constant for a specific sensor (`Sensor.TYPE_ACCELEROMETER`).

Similar to sensor APIs in Android, raw sensor data is monitored by implementing the `onAccuracyChanged()` and `onSensorChanged()` callback methods provided through the `ShareSensListener` interface.

```
public class ShareSensExample implements
     ShareSensListener {
  private ShareSensManager mShareSensManager;
  private Sensor mAccelerometer;

  @Override
  public ShareSensExample() {
   mShareSensManager = (ShareSensManager)
     getSystemService(Context.SENSOR_SERVICE);
    mAccelerometer = mShareSensManager.
     getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    mShareSensManager.registerListener
      (this, mAccelerometer,
        ShareSensManager.SENSOR_DELAY_NORMAL);
  }

  @Override
  public final void onSensorChanged(SensorEvent
           event) {
    // Do something with the sensor event.
  }

  @Override
  public final void onAccuracyChanged(
     Sensor sensor, int accuracy) {
    // Do something here if sensor
    // accuracy changes.
  }
}
```

Figure 4. ShareSens API Usage Example

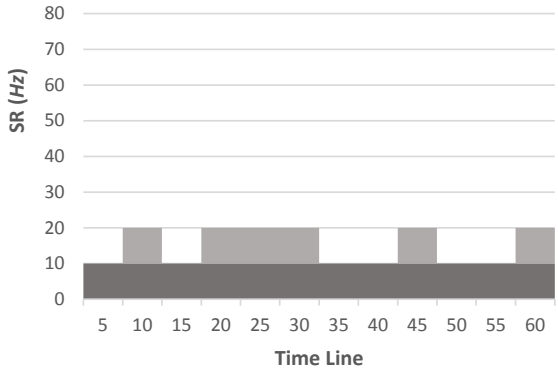
V. Evaluation

A set of experiments was carried out to measure the overall improvement achieved in power consumption by using ShareSens, as well as ShareSens's power footprint. We used the PowerTutor software [28] for our power measurements.

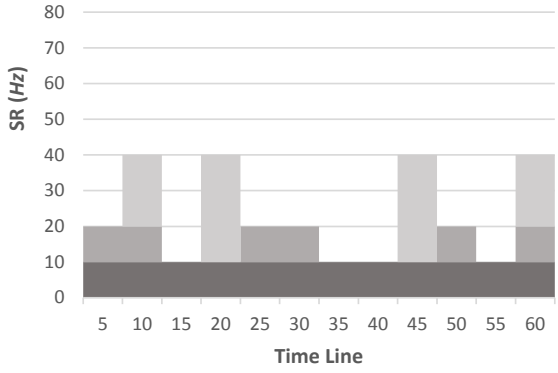
A. Experimental Setup

For our experiments, we used a Samsung Galaxy Note II phone running Android OS ver 4.4, and compared ShareSens with the default Android API for the power consumed in serving identical sensing requests.

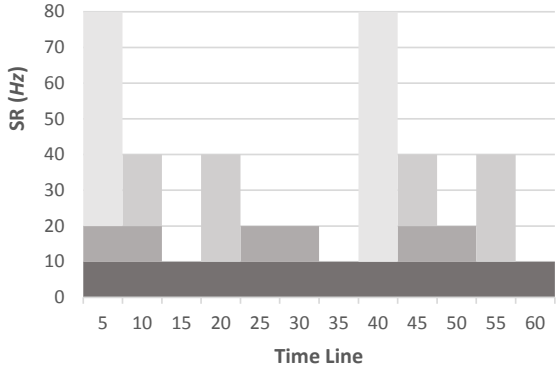
To measure the overall improvement in power consumption, we made measurements of power used by ShareSens and the Android API for artificially generated sensing loads. The loads were designed to simulate applications with mixes of continuous sensing demands, ranging from light to heavy. For each case, measurement were done over the period of 60 seconds, where fresh requests could arrive at 5 second points. In all cases, there was a 10 Hz request which lasted the entire 60 seconds. Additionally, 20 Hz, 40 Hz, and 80 Hz requests were inserted at randomly selected points, each for 5 seconds at a time. The maximum number of requests of a certain sampling rate were limited depending on the frequency: 6 of 20 Hz, 4 of 40 Hz, and 2 of 80 Hz. This was to simulate the likelihood that higher sampling demands – such as for fine-grained health monitoring requiring 80 samples per second – would be less frequent. The same three sets of measurements were carried out for two different sensors, the accelerometer and the orientation sensor. Figure 5 shows the light, medium and heavy loads used in the experiments.



(a) Light Load



(b) Normal Load

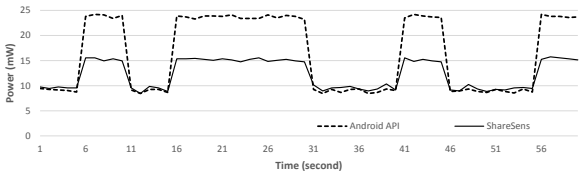


(c) Heavy Load

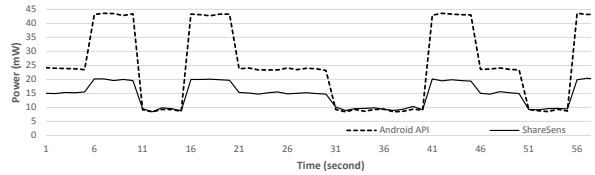
Figure 5. Experimental Setup: Sensing Workload Scenarios

B. Power Consumption of ShareSens Vs Sensor APIs in Android

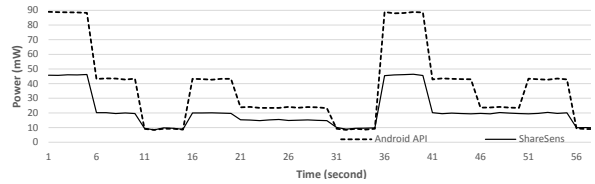
As Figures 6 and 7 show for the accelerometer and orientation sensor experiments, ShareSens delivered the power savings we expected it to, effectively reducing the power use to the requirement for the highest sampling rate being requested at any time, independently of the



(a) Power Consumption of Accelerometer in Light Load

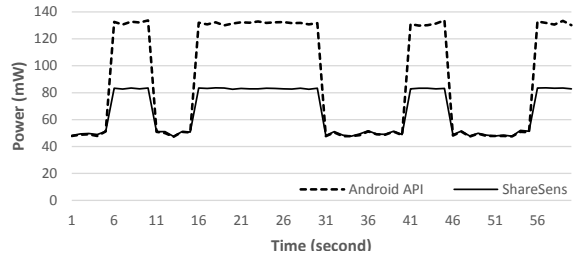


(b) Power Consumption of Accelerometer in Normal Load

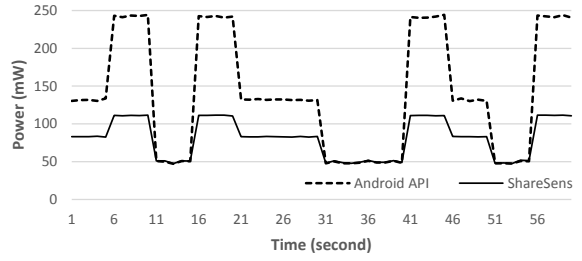


(c) Power Consumption of Accelerometer in Heavy Load

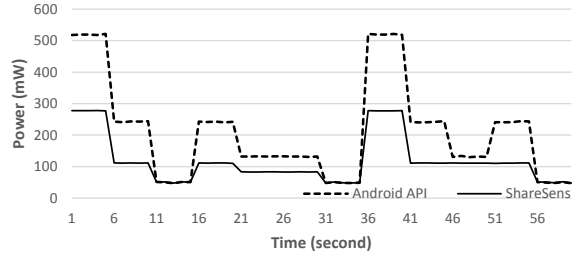
Figure 6. Power Use Measurements for Accelerometer



(a) Power Consumption of Orientation in Light Load



(b) Power Consumption of Orientation in Normal Load



(c) Power Consumption of Orientation in Heavy Load

Figure 7. Power Use Measurements for Orientation Sensor

additional requests for lower sampling rates. This was partly because the sampling demands we chose for the sensing tasks were of the form $10 * 2^n$ Hz – somewhat comparable to the sampling frequencies that Android's API expects applications to typically pick from (5Hz, 16Hz, 50Hz, 100Hz) – and the lower sampling frequencies could be served out of the higher sampling frequencies. More generally, the power required would depend on the GCD of the reciprocal of the time between individual samples taken.

Surprisingly, using the Android API led to power consumption that was superlinear with respect to the sum of all sampling rates requested by the concurrent sensing tasks; most notably, in our tests, we found that multiple requests adding up to a cumulative sampling rate of (say) 80Hz used less power than a single request for 80Hz, leading to a negative correlation between the number of independent sensing streams and the power required to serve them, for the small number of streams we tested for.

Table I shows the total amount of energy used for the entire 60 seconds sensing loads used in our experiments. The percentage saving in the energy consumed when using ShareSens (in comparison with the Android API) depends entirely on the number of requests being served, because although the power used is roughly linear in the cumulative sampling rate of all requests for the Android API, for ShareSens, it depends almost entirely on the highest frequency being requested at the time.

Table I
POWER CONSUMED FOR DIFFERENT WORKLOADS IN MJ

Sensor	Workload	Android		ShareSens	
		Avg.	SD	Avg.	SD
Accelerometer	Light	980	49	739	38
Accelerometer	Normal	1,368	82	955	44
Accelerometer	Heavy	1,822	98	1,249	70
Orientation	Light	5,427	282	3,982	239
Orientation	Normal	8,464	406	4,879	234
Orientation	Heavy	12,752	587	7,127	356

C. Overhead Analysis

In order to determine the non-sensing overhead of ShareSens, we measured the energy consumed for the various sensing loads presented previously, albeit without the actual sensing. For the light, normal, and heavy workloads, the energy consumed was 27 mJ, 38 mJ and

52 mJ, respectively, for the accelometer, and very similar 30 mJ, 39 mJ and 54 mJ for the orientation sensor. In percentage terms, this was roughly 4% of the total energy consumed in the accelerometer experiments, and 0.8% for the orientation sensor, the difference explained by the order-of-magnitude larger overall energy demand of the orientation sensor itself.

VI. Conclusions

With the growing ubiquity of sensor-equipped smartphones, wearables, and sensors in the environment, it is more possible than ever to offer innovative applications which utilize data that can be collected through these sensors. However, these applications require continuous sensing, and as the number of applications grows, so will the demand on sensors, and the consequent demand on batteries powering these devices. We have addressed this challenge by developing an approach for optimizing the frequency of data collection by sensors, which takes advantage of the overlap in sensing requirements of various applications. We presented our design and prototype implementation of the ShareSens API for Android, which can be used by applications to request access to sensor data. Our experimental evaluation shows that significant power savings can be achieved using ShareSens when there are overlapping sensing requests.

In future work, we are looking into opportunities for updating existing applications to use ShareSens. Although a tool could easily be developed to automatically change source code to use the ShareSens API instead of the default Android API, it is not obvious that this would lead to the best outcomes. Particularly, programmers coding while assuming ShareSens's properties may very well choose different sampling rates for the various sensors from what they would choose when working with the default Android API. One direction we hope to explore is the ability to allow programmers to opportunistically pick higher sampling rates when already available at a low marginal cost. This would require a decision mechanism based on querying for the current sensor sampling rates, and adapting to it. A related question concerns fairness among different applications in terms of which applications can request how much sampling, and consequently use how much energy, and how to account for this when multiple applications piggyback on each other's sampling rates. One possibility would be to treat sampling privileges as precious resources, and use an approach based on our CyberOrgs[6] model [27] to coordinate competition for them.

[6]CyberOrgs is a model for resource-bounded multi-agent computations.

Acknowledgment

Support from the Natural Science and Engineering Research Council of Canada (NSERC), the Canadian Foundation for Innovation (CFI) and the Government of Saskatchewan is gratefully acknowledged.

References

[1] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. Campbell, "A survey of mobile phone sensing," *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 140–150, September 2010.

[2] A. Abdel Moamen and N. Jamali, "Coordinating crowdsourced services," in *Proceedings of the Third IEEE International Conference on Mobile Services*, Alaska, USA, June 2014, pp. 92–99.

[3] N. Vallina-Rodriguez and J. Crowcroft, "Energy management techniques in modern mobile handsets," *Communications Surveys Tutorials, IEEE*, vol. 15, no. 1, pp. 179–198, Jan 2013.

[4] L. Besaleva and A. Weaver, "Crowdhelp: A crowdsourcing application for improving disaster management," in *Proceedings of the 2012 IEEE Conference on Global Humanitarian Technology*, ser. GHTC '13, 2013, pp. 185–190.

[5] F. Ye, R. Ganti, R. Dimaghani, K. Grueneberg, and S. Calo, "Meca: Mobile edge capture and analysis middleware for social sensing applications," in *Proceedings of the 2012 International Conference Companion on World Wide Web*, ser. WWW '12 Companion. ACM, 2012, pp. 699–702.

[6] M. Zook, M. Graham, T. Shelton, and S. Gorman, "Volunteered geographic information and crowdsourcing disaster relief: A case study of the haitian earthquake," *World Medical and Health Policy*, vol. 2, no. 2, pp. 7–33, 2010.

[7] R.-Y. Li, S. Liang, D.-W. Lee, and Y.-J. Byon, "Trafficpulse: A mobile gisystem for transportation," in *Proceedings of the 2012 ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems*, ser. MobiGIS '12. New York, NY, USA: ACM, 2012, pp. 9–16.

[8] U. Levine, A. Shinar, E. Shabtai, and Y. Shmuelevitz, "Condition-based activation, shut-down and management of applications of mobile devices," *United States Patents, US 8,271,057*, 2009.

[9] S. Nawaz, C. Efstratiou, and C. Mascolo, "Parksense: A smartphone based sensing system for on-street parking," in *Proceedings of the 2013 ACM International Conference on Mobile Computing Networking*, ser. MobiCom '13. New York, NY, USA: ACM, 2013, pp. 75–86.

[10] H. Martín, A. M. Bernardos, J. Iglesias, and J. R. Casar, "Activity logging using lightweight classification techniques in mobile devices," *Personal Ubiquitous Comput.*, vol. 17, no. 4, pp. 675–695, 2013.

[11] G. Bieber, T. Kirste, and M. Gaede, "Low sampling rate for physical activity recognition," in *Proceedings of the 7th International Conference on PErvasive Technologies Related to Assistive Environments*, ser. PETRA '14, 2014, pp. 15:1–15:8.

[12] Y.-S. Lee and S.-B. Cho, "Activity recognition using hierarchical hidden markov models on a smartphone with 3d accelerometer," in *Proceedings of the 6th International Conference on Hybrid Artificial Intelligent Systems - Volume Part I*, ser. HAIS'11, 2011, pp. 460–467.

[13] T. Uga, T. Nagaosa, and D. Kawashima, "An emergency earthquake warning system using mobile terminals with a built-in accelerometer," in *Proceedings of the 2012 IEEE Conference on ITS Telecommunications*, ser. ITST '12, November 2012, pp. 837–842.

[14] C. Hung-Ching, A. Agrawal, and K. Cameron, "Energy-aware computing for android platforms," in *Proceedings of the 2011 International Conference on Energy Aware Computing*, ser. ICEAC '11, Nov 2011, pp. 1–4.

[15] B. Dietrich and S. Chakraborty, "Managing power for closed-source android os games by lightweight graphics instrumentation," in *Proceedings of the 12th Annual Workshop on Network and Systems Support for Games*, ser. NetGames '12, 2012, pp. 1–3.

[16] U. Y. Ogras, R. Z. Ayoub, M. Kishinevsky, and D. Kadjo, "Managing mobile platform power," in *Proceedings of the 2013 International Conference on Computer-Aided Design*, ser. ICCAD '13, 2013, pp. 161–162.

[17] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, no. 4, pp. 51 –56, april 2010.

[18] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, ser. MobiSys '10, 2010, pp. 49–62.

[19] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh, "A framework of energy efficient mobile sensing for automatic user state recognition," in *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '09, 2009, pp. 179–192.

[20] H. Lu, A. J. B. Brush, B. Priyantha, A. K. Karlson, and J. Liu, "Speakersense: Energy efficient unobtrusive speaker identification on mobile phones," in *Proceedings of the 9th International Conference on Pervasive Computing*, ser. Pervasive'11, 2011, pp. 188–205.

[21] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "Sociablesense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '11, 2011, pp. 73–84.

[22] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh, "A framework of energy efficient mobile sensing for automatic user state recognition," in *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '09, 2009, pp. 179–192.

[23] K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, and M. Welsh, "Mercury: A wearable sensor network platform for high-fidelity motion analysis," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09, 2009, pp. 183–196.

[24] B. Priyantha, D. Lymberopoulos, and J. Liu, "Littlerock: Enabling energy-efficient continuous sensing on mobile phones," *Pervasive Computing, IEEE*, vol. 10, no. 2, pp. 12–15, April 2011.

[25] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[26] M. Jang, A. Momen, and G. Agha, "Efficient agent communication in multi-agent systems," in *Proceedings of the 2004 Software Engineering for Multi-Agent Systems*, ser. SELMAS '04, 2004, pp. 236–253.

[27] N. Jamali and X. Zhao, "Hierarchical resource usage coordination for large-scale multi-agent systems," in *Massively Multi-Agent Systems I.* Springer Berlin Heidelberg, 2005, pp. 40–54.

[28] Powertutor: A power monitor for android-based mobile platforms. [Online]. Available: http://ziyang.eecs.umich.edu/projects/powertutor/