A Semantics-Aware Design for Mounting Remote Sensors on Mobile Systems

Yu-Wen Jong[1], Pi-Cheng Hsiu[2], Sheng-Wei Cheng[1], and Tei-Wei Kuo[1,2]
[1] Department of Computer Science and Information Engineering, National Taiwan University, Taiwan
[2] Research Center for Information Technology Innovation, Academia Sinica, Taiwan
r02922053@csie.ntu.edu.tw, pchsiu@citi.sinica.edu.tw, {d02922004, ktw}@csie.ntu.edu.tw

ABSTRACT

Application paradigms will increasingly exceed a mobile device's physical boundaries. This paper presents a system solution for a mobile device to mount remote sensors on other devices. Our design is generic to mobile senor stacks, thus supporting unmodified apps and commodity sensors. Furthermore, it uses an asynchronous access model to facilitate semantics passing and data reporting in between. Such semantic information allows the development of an energy-efficient reporting policy for remote sensing applications. The results of experiments conducted on commercial Android smartphones with popular apps demonstrate that our design is very efficient in terms of energy consumption and completion time.

CCS Concepts

•**Computer systems organization** → **Embedded systems;** *Embedded software;* •**Software and its engineering** → **Operating systems;** *Communications management;*

Keywords

Mobile sensing, remote access, energy efficiency, mobile systems

1. INTRODUCTION

Until recently, most mobile apps were intended for standalone platforms. Soon, many users may own multiple mobile devices, such as smartphones, tablets, and wearables, which relate to one another in different ways. In this trend, many apps provide a consistent user experience across platforms; for example, a user can watch video steams on both his smartphone and tablet. This *consistent paradigm* allows a user to access the same application or service via any device. Some apps use the *continuous paradigm*, where the experience can flow from one platform to another. For instance, a user can start composing an email on his tablet, and then pick up where he left off on his smartphone. However, these paradigms do not leverage the distinctive features of different devices. For example, a smartphone with a powerful hardware specification may be more suitable for computation, whereas a smartwatch worn on a user's wrist is better at collecting information, making each device type better suited for particular tasks. Thus, users can share their smartwatches' motion sensors with friends' smartphones to compare and help improve their daily exercise routines. Taking advantage of these differences leads to the *complementary paradigm*, which enables the development of new applications which are best used with multiple devices [10]. This observation motivates us to design a unified access model for sharing various sensors between mobile systems, thereby allowing mobile apps to exceed a device's physical boundaries.

Solutions for sharing I/O devices already exist. For example, IP Webcam [15] exposes a smartphone's camera so that it can be viewed from another smartphone; SoundSeeder [6] allows for music to stream to other smartphones, turning them into wireless speakers. The emergence of smart wearables have also sparked some interesting use cases, where an app running on a smartphone has to access certain sensors on a wearable device. Sleep as Android [20] is a smartphone app which reads motion sensors on a smartwatch to detect the user's sleep patterns, and Cinch [19] accesses the smartwatch heart sensor to track the user's heart rate. Various *cooperative sensing* applications that exchange sensor data with registered or nearby mobile devices have also emerged, e.g., PRISM [3], CoMon [9], and Remora [8]. These solutions are application-specific; in other words, the features are hard-coded inside the apps and cannot be reused by other apps. To facilitate the development of multi-device apps, Google and Samsung have recently released remote sensing supports, in the form of new Android APIs, called Google Fit [4] and Samsung Remote Sensor [17] respectively, intended for their own wearable devices. However, these application-level solutions all have a fundamental limitation, namely *portability*, in that they do not support existing apps without modifications.

To sustain unmodified apps, the remote sensing support must be done at a lower layer, e.g., the kernel layer. One well-known example of kernel-layer I/O sharing solutions is remote file systems like NFS [18], which allows a device to access remote files over a network much in the way that local files are accessed. Similarly, Rio [1] uses *device files*, as they are transparent to the application layer, to abstract various I/O devices; then, it can directly forward all file operations intercepted on one mobile system hosting the app to another equipped with the I/O device. However, the kernel-layer solution achieves device-agnostic I/O sharing at a cost of considerable communication overhead because every operation turns into one round-trip transmission. This is especially a concern for mobile sensing because such applications are typically implemented as background services that collect data continuously. To reduce communication energy cost, ACQUA [13] provides a *context-aware middleware* for upper-layer apps to selectively pull only relevant data from external sensors, while SeeMon [7] only transmits data with significant changes. In other words, they reduce the volume of transmitted sensor data by exploring application context, provided that they are capable of sensor-specific query processing and data filtering. Thus, these middle-tier solutions cannot *transparently* access any sensors, requiring significant engineering effort to support new commodity sensors.

In this paper, we present a system solution, which is very efficient in terms of both communication energy and time costs, for sharing various sensors between mobile systems. Our solution overcomes the limitations of application-level and kernel-layer solutions by adding our remote support at the *hardware abstraction layer*, a layer low enough to allow for transparency and portability but high enough to carry semantic information.

DAC '16, June 05-09, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897937.2897975

First, we realize a remote sensor stack by splitting the local sensor I/O stack between two mobile systems; through an established wireless connection, each software *sensor stub* that abstracts a hardware sensor device on one system interacts with the remote *sensor daemon* which manages all enabled remote sensors on the other system. Then, to save communication energy, we replace the inherently synchronous access model, where data is accessed through the use of sequentially issued operations, with our *asynchronous* protocol, where nearly all wireless transmissions are used for data reporting. Next, we propose a reporting policy that exploits semantic information to aggressively preserve a mobile system's sleep pattern, as if no remote sensors were used, so as to avoid extra wakeup energy costs. Finally, to validate our design's efficacy, we integrated it into Android and conducted extensive experiments on LG Nexus 5 and Samsung S3 smartphones with popular mobile apps from Google Play. Compared to a straightforward extension of the local synchronous model, in a range of different scenarios, our design on average reduces energy consumption of a mobile system that uses remote sensors by 32.4%, while improving completion time of each data reporting by 81.8%. We hope our design serves as an underlying interconnection mechanism among heterogeneous mobile devices to facilitate multi-device applications.

The remainder of this paper is organized as follows. Section 2 provides background information and explains the motivation for this work. In Section 3, we present the philosophy and details of our semantics-aware design. The experiment results are reported in Section 4. Section 5 presents some concluding remarks.

2. BACKGROUND AND MOTIVATION

2.1 Mobile Sensors

Modern mobile devices are equipped with various sensors which are used in various ways by different applications. Mobile sensors collect various kinds of information and generate data in different *reporting modes*. For example, Android supports three reporting modes: *one-shot*, *on-change*, and *non-stop*. A one-shot sensor, like a pick-up gesture sensor, generates data only once, while an on-change sensor, such as a proximity sensor, generates data whenever there is a significant change in value. Examples of non-stop sensors include gyroscopes, accelerometers, and magnetometers, which generate data continuously at a constant rate. Most sensors used in mobile apps are non-stop sensors [12], which allow application developers to adapt *data rates* to their needs. Once a sensor generates data, the mobile system is notified to issue operations to read the collected data from the sensor. Such a procedure is called one *reporting*.

To preserve energy, a mobile system following an *aggressive sleeping philosophy* [14] stays asleep by default unless awakened passively by some application (or system) program. In consequence, the mobile system awakens from a sleep whenever a one-shot or an on-change sensor reports. However, continuous reporting from non-stop sensors may force the mobile system to stay awake all the time to ensure no data is lost. To alleviate this problem and provide increased opportunities for the system to sleep, mobile sensors now support *hardware batching* [16] through sensor hubs to buffer data for a period of time before reporting the data. In Android, app developers can set the maximum latency to specify how long the data can be delayed before reporting; this latency is essentially the *report period* for a non-stop sensor because it determines how often the sensor reports data to the system. Generally, the larger the report period, the more energy can be saved because the less frequently the mobile system needs to wake up. However, sensors have limited buffer space and, once the buffer is filled up, the mobile system will still be awakened before the report period is up to prevent data loss (unless the sensor is designated as a non-wakeup sensor that allows data loss).

2.2 From Local Sensing to Remote Sensing

Remote sensing enables information augmentation by extending local sensing. Henceforth a mobile app running on a *client* (e.g., a smartphone) can use the sensing data collected by a *server* (e.g., a smartwatch or another smartphone). Accordingly, one reporting consists of the sensor generating data on the server side, the mobile system then being notified on the client side, and finally the client's system issuing operations to read a batch of data from the server's sensor. To realize remote sensing, the sensor I/O stack is typically split between two mobile systems, known as a *split-stack architecture* [1]. Splitting the stack at different layers leads to different operations being transmitted in between. For instance, if the stack is split at the application layer, *remote procedure calls* defined by app developers cause message passing between the client and the server to complete each reporting. By contrast, if the stack is split at the kernel layer for generic applications, *system calls* (such as `open`, `ioctl`, and `read` provided in Linux) are forwarded in sequence to be executed on the server and interleaved with the execution results returned to the client. Figure 1 illustrates the interaction between the two entities. In essence, all operations intended for a sensor need to be forwarded from the client to the server and executed there instead; as a result, every operation will turn into one round-trip transmission [1].

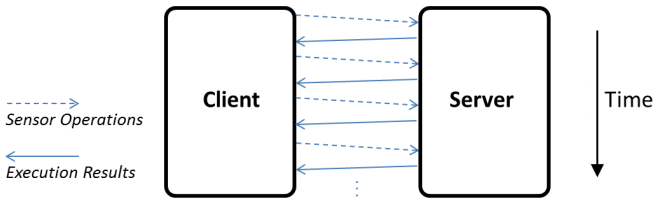


Figure 1: Synchronous access model

Issuing operations to a *local* sensor for data is a sensible design because it only involves normal function invocations. However, acquiring data from a remote sensor through issuing operations incurs additional network transmissions. In other words, the energy consumed for one reporting is largely increased. Although the problem of energy consumption on the server side can be mitigated by lengthening the report periods of sensors (and increasing the buffer size in hardware batching if necessary), the problem becomes especially severe for the client as it can be associated with multiple remote sensors, thereby forming a one-to-many relationship. More specifically, continuous sensor reporting may result in the client being awakened individually by different remote sensors, thus making the client very energy hungry when using remote sensing applications. The cost incurred when accessing data through issuing operations to a remote sensor raises the need for a remote access model that reduces the energy required for every reporting without incurring data loss.

3. SEMANTICS-AWARE REMOTE SENSING

Our design exploits sensor semantics and system states to achieve energy-efficient remote sensing. In Section 3.1, we describe an asynchronous access model to facilitate semantics passing and data reporting across the two systems. Based on the model, we then propose an energy-efficient reporting policy in Section 3.2.

3.1 Asynchronous Access Model

3.1.1 Remote Sensor Stack

We extend the local sensor I/O stack to add our remote sensing support. In the local stack, applications access sensor devices through a *hardware abstraction layer*, called the sensor HAL, which provides APIs to the upper-layer applications and uses function calls provided by lower-layer device drivers. Under

the HAL, every software sensor stub represents one hardware sensor device (even though the physical hardware sensor may not exist). This layer is low enough to hide differences in sensor hardware from the mobile system and applications, but high enough to be aware of sensor semantics (e.g., reporting modes and periods) and system states (e.g., awake or asleep). Moreover, realizing remote sensing at this layer with well-defined interfaces eliminates the need to modify apps and device drivers, thereby achieving both transparency and portability.

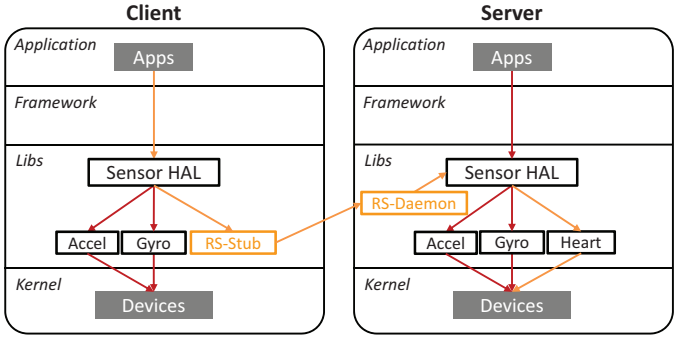


Figure 2: Our remote sensor I/O stack

Figure 2 shows our remote sensor stack. We add *remote sensor stubs* (rs-stubs), each of which represents a remote sensor, under the HAL. Like local sensor stubs, an rs-stub implements the essential interfaces (e.g., `enabling`, `disabling`, and `read`) defined by the HAL, so it is viewed as a regular sensor by the system and all apps. However, instead of communicating with an underlying hardware sensor, it communicates with the *remote sensor daemon* (rs-daemon), which manages all enabled remote sensors, above the HAL on the server side. Specifically, on receipt of the sensor operations issued by the HAL (as a consequence of the operations issued from the upper layers), the rs-stub does not act like a local stub that issues its operations to read the data from the sensor device into its own buffer; instead, it interacts with the rs-daemon via an established wireless connection according to our access protocol presented below.

3.1.2 Access Protocol

An intuitive protocol is to directly forward every operation of an rs-stub and wait for the rs-daemon to return the result of executing the operation. However, such a *synchronous* protocol, which simply forwards operations and results in an alternating manner, consumes significant amounts of energy for communication. To reduce energy overhead, we make the communication *asynchronous*; that is, the server starts (or stops) reporting the data pro-actively once the client enables (or disables) the remote sensor. The communication model in Figure 1 is thus simplified, as shown in Figure 3, where (aside from a few enabling, disabling, and updating messages) all wireless transmissions are intended for data reporting. Moreover, the addition of our remote sensing support does not affect the synchronous access model originally designed for local sensing.

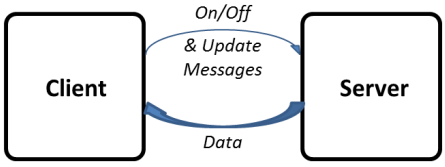


Figure 3: Our access protocol

To enable a remote sensor, the corresponding rs-stub combines the enabling operation with semantic information (i.e., the reporting mode, data rate, and report period) into a single request and sends the request to the rs-daemon. Then, the rs-stub launches a *receiver thread*, which will keep receiving data from the rs-daemon and putting the data into the rs-stub's buffer, as well as notifying a system service (called `SensorService` in Android's framework), until the sensor is disabled via a disabling request. Whenever the system service is noticed, it will read the data (as if from a local sensor stub) into its buffer and then distribute the data to the apps that are listening to the sensor. Afterwards, a new request is sent only when the semantic information is updated; otherwise, the client remains passive. Apart from the semantic information, the client also shares its state information (i.e., awakening or sleeping along with the next wakeup time), so that the server can decide when to report data. Once a state change occurs, the client sends a sleeping or awakening message to the rs-daemon. State changes can be captured by intercepting two respective functions (called `suspend_prepare` and `backlight_on` in the Android kernel) invoked when the system is awakening and falling asleep. Moreover, before falling asleep, the client is aware of when it will wake up again. This information is available because, for a mobile system which stays asleep by default, it must provide an alarm mechanism (called `AlarmManager` in the Android framework) so that apps can register the time points at which the system should awake to execute their tasks. The next wakeup time is also carried in the sleeping message.

For each remote sensor enabled, the rs-daemon creates a *transmitter thread*, which will keep reading the data generated by the sensor device through the HAL (exactly like accessing a local sensor) and transmitting the data to the rs-stub until the sensor is disabled by the client. Whenever the transmitter thread is noticed by the sensor device, it will read the data from the corresponding sensor stub into its own dynamically-expanding buffer[1], ensuring that all data is stored for later reporting. During the interaction, the server assumes an active role to report data, except that it may be passively woken up by the client's occasional messages. The rs-daemon also employs a *coordinator thread*, which is responsible for receiving the sleeping and awakening messages from the client, as well as deciding the reporting times for all enabled sensors according to our reporting policy presented below.

3.2 Energy-Efficient Reporting Policy

3.2.1 One-to-One Access

A straightforward policy is to perform reporting in accordance with each sensor's report period; however, this may require the client to constantly wake up and activate its network interface for data reception, especially when multiple remote sensors are simultaneously enabled. Hence, we propose an energy-efficient reporting policy inspired by an observation on the behavior presented by sensors and that expected by users. When the client is asleep, whether apps receive data in line with the report periods makes no difference in terms of the user experience, making data reporting delay-tolerant. In contrast, when the user is interacting with the client, it must obey all app-specified settings. Accordingly, the principle behind our policy is to aggressively maintain the client's sleep pattern as if no remote sensors were used, while preventing data loss. To this end, the server delays the data reporting of non-stop sensors until the client awakens from sleep, because they typically collect data for background services when the screen is turned off. However, one-shot and on-change sensors are allowed to make instant reports, because their data is generated occasionally and may require timely delivery. The server reports data in two ways, depending on whether the client is wakened by a scheduled or unscheduled event.

Scheduled Wakeups: A mobile system, while left unused,

[1]If the buffer cannot be further expanded or has reached its maximum size, the transmitter thread has to awaken the client by directly transmitting the accumulated data to the rs-stub.

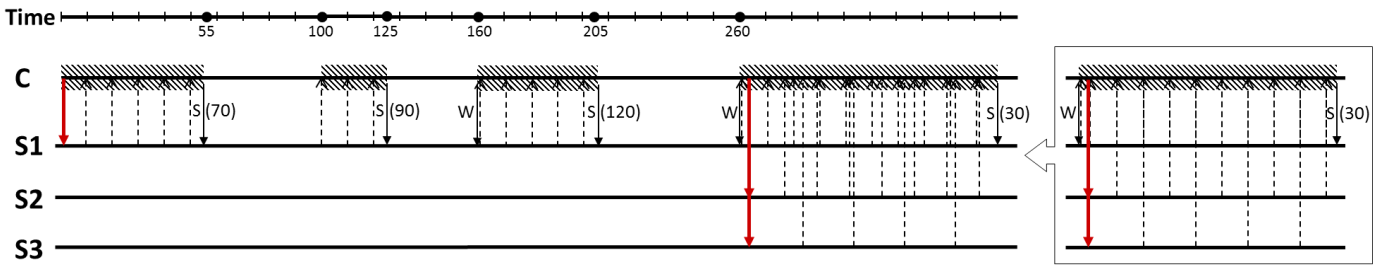


Figure 4: Example to illustrate our reporting policy

still frequently wakes up for short periods to execute tasks (most of which involve network communications) registered *a priori* by internal apps or system services [11]. Reporting data during these scheduled wakeups imposes negligible energy costs compared with the energy required to activate the whole system. To leverage this, right before going into sleep, the client sends out a sleeping message (carrying its next wakeup time), as shown at $t = 55$ in Figure 4. On receipt of the message, the server stops reporting the data of those sensors that it considers delay-tolerant, and schedules the next reporting to catch up the client's next *short* wakeup. During the sleep period, all data generated by those sensors are stored in the rs-daemon's buffer. At the next reporting time, which is $t = 100$ in the example, the rs-daemon automatically batch reports all the data accumulated since the last reporting.

Unscheduled Wakeups: A mobile system can also be wakened unexpectedly by an external event. One important case[2] is when the user turns on the screen and starts to interact with the device. Once the user presses the power button, the client sends out an awakening message to acquire the data accumulated at the server, so as to avoid potential negative impact on the user experience. Once the server receives the message, at $t = 160$ in the example, the rs-daemon reports all the accumulated data immediately, and continues to report new data at the interval of the report period until the server receives a sleeping message at $t = 205$.

By reporting data only during wakeups caused by other events, remote sensing only imposes limited energy costs, although the client would be forced to awake for slightly longer than the original wakeups depending on the data batch sizes. Moreover, by reporting data continuously when the user is interacting with the client, the application behavior remains unchanged from the user's perspective. The design concept could be applied to different wireless interfaces, but a reliable connection like TCP should be established between the client and server to ensure successful and in-order delivery of all data and messages.

3.2.2 One-to-Many & Many-to-Many Access

Some scenarios rely on one-to-may access, where a client mounts multiple remote sensors on different servers. Like one-to-one access, one-to-many access also delays the reporting of non-stop sensors by ignoring their report periods when the client is asleep, but obeys this parameter when the client stays awake for a while. However, due to the different report periods of remote sensors, the client may have to constantly turn on and off its network interface for data reception, resulting in unnecessary energy consumption during switching. For example, in Figure 4, with two additional remote sensors enabled at $t = 260$, data reporting becomes scattered and frequent. To solve this problem, the client aligns the report times of all the enabled sensors before it sends enabling requests to the servers. One simple adjustment is to *slightly shorten* all report periods such that they are all multiples of the smallest one. For example, suppose that the respective report periods of S1, S2, and S3 are 10, 12, and 21; the client slightly adjusts them to 10, 10, and 20 respectively so that they are all multiples of the smallest period. Note that to preserve the user's expectation of the application behavior, a report period can be reduced to its closest smaller value, but not expanded to a bigger value. By aligning the report times, the client need not frequently turn on and off its network interface, as shown in the right box of Figure 4.

If multiple clients mount an identical remote sensor with different parameter settings, forming many-to-one access, then the server simply reports individually to each awakened client, just like the case when multiple apps listen to the same sensor in local sensing. Lastly, the above polices can directly combine to form the policy for many-to-many access.

[2]Another case is when some network packets sent by external facilities arrive. We disregard this case because the resulting wakeups are typically short and unpredictable.

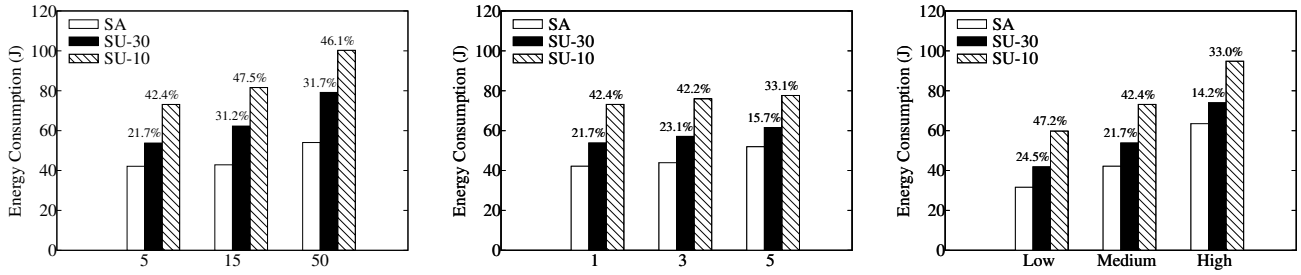
4. PERFORMANCE EVALUATION

4.1 Experiment Setup

We realized our design by hacking an open source sensor HAL released by AKM [2] and replaced the native HAL in Android (version 4.4) with ours. To evaluate the design, we conducted extensive experiments on commercial smartphones, with an LG Nexus 5 (serving as the client) accessing remote sensors residing on two Samsung Galaxy S3 (playing the role of servers). The smartphones communicated over a TCP connection via a TP-Link 802.11n access point dedicated for our experiments. We used the Monsoon Solutions power monitor to measure the client's energy consumption. In addition, we inserted the `gettimeofday()` function provided by Linux into every rs-stub to measure the completion time of each data reporting. Note that the reported results could also reflect the results of multiple clients accessing multiple remote sensors, because the servers simply report individually to each client in many-to-many access.

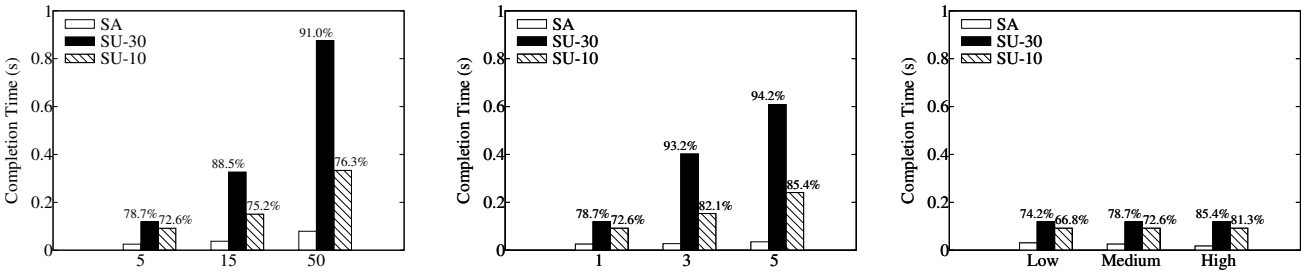
The client ran an open source app called SensorList [5], which allows us to parameterize and use any sensors available on a mobile system. We used common non-stop sensors including a gyroscope, an accelerometer, and a magnetometer, along with light and pressure sensors. To reduce the complexity of the experiments, we set the same data rate and report period for all used sensors to produce consistent behavior, and evenly distributed them between the two servers at random. Moreover, data reporting piggybacks onto innate wakeups, but whether the wakeups are scheduled or unscheduled makes no significant difference. To reduce the potential impact of human intervention, we generated (scheduled) wakeups by installing real-world apps on the client. The wakeup frequency was deemed low when the client was recovered to its factory settings. For the medium frequency, we installed the full suite of Google OpenGApps. The high frequency included six additional popular apps, namely Line, WhatsApp, Facebook, Facebook Messenger, Yahoo Mail, and Yahoo Messenger.

We investigated the impacts of different factors on our design, including: 1) the impact of the data rate at 5, 15, and 50 Hz, 2) the impact of low, medium, and high wakeup frequency, and 3) the impact of 1, 3, and 5 remote sensors. Unless otherwise specified, the default data rate was 5 Hz, which is also the default value used in Android. We considered the medium frequency



(a) Sensor data rate (b) Number of sensors (c) System wakeup frequency

Figure 5: Energy consumed by SA and SU under different factors



(a) Sensor data rate (b) Number of sensors (c) System wakeup frequency

Figure 6: Completion time achieved by SA and SU under different factors

and one remote sensor by default because they might be normal use cases. In addition, to better understand the compound impacts of these factors in daily use, we emulated two usage scenarios. First, we considered a user who uses a lifelogging app to record his daily activities (e.g. sleeping or exercising). In this scenario, two sensors with a default data rate of 5 Hz are sufficient to detect the user's activities. We assumed this user had installed many apps on his smartphone, resulting in a high wakeup frequency. Next, we considered a user who uses a posture correction app in a yoga class to track her arm and leg movements. In this case, five sensors with a data rate of 50 Hz are needed to capture fine-grained motions; we assumed that the user does not have many apps installed and her smartphone has a low wakeup frequency.

We compared our semantics-aware design, denoted as SA, with a semantics-unaware straightforward extension of the local access model, denoted as SU. Like Rio [1], which implements the split-stack architecture at the kernel layer, SU splits the sensor I/O stack at the HAL layer and directly forwards all sensor operations and the results in between (as described in Section 2.2). In SU, when to report depends entirely on the report periods specified by the apps. However, we observed that most sensor apps do not specify this (optional) parameter, leaving it with the default value 0 used in Android. This, in turn, leaves the mobile system permanently awake and extremely energy-hungry. For a more balanced comparison, we picked SU with report periods set at 10 and 30 seconds, denoted respectively as SU-10 and SU-30. To show the efficacy, the adopted metric was the total energy (in joules) consumed by the client to perform remote sensing for 15 minutes (which was long enough to reveal some regular patterns). We also measured the average time (in seconds) required to complete one reporting (or, more precisely, the difference between the start and end times an rs-stub receives a batch of remote data).

4.2 Energy Consumption

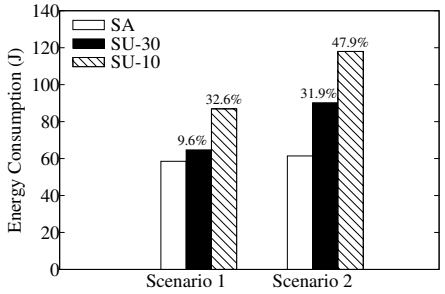
Figure 5 shows the energy consumed by SA, SU-30, and SU-10 under three different factors. As expected, the energy consumption increases with the sensor data rate, number of sensors, and system wakeup frequency. The data rate and the number of sensors respectively determine the batch size in each reporting and the reporting times; while the system wakeup frequency reflects the extra energy required for scheduled wakeups. Naturally, the energy consumption increases with the three factors. However, as the data rate grows, the energy consumption increases more quickly under SU-10 and SU-30 than under SA, as shown in Figure 5(a). This is because SU relies on back-and-forth operations to acquire remote data; thus, the more data, the faster the increase in energy consumption. The same phenomenon occurs with the growing number of sensors in Figure 5(b). Note that the servers' report times (which implies the client's wakeup times) are aligned by setting the same report period for all sensors; otherwise, the increase under SU will be more drastic. As shown in Figure 5(c), with the increasing wakeup frequency, the increase in energy consumed by SA is clearly greater than in Figures 5(a) and 5(b). This increased energy consumption is mainly due to scheduled wakeups, rather than remote sensing. This also explains why the increments under SA and SU are similar in Figure 5(c). Overall, SA can reduce the energy consumption of SU-30 by between 14.2% and 31.2%, and that of SU-10 by between 33% and 47.5%. This large energy reduction is due to SA striving to maintain the client's original sleep pattern, so that energy is consumed mainly for data reception.

4.3 Completion Time

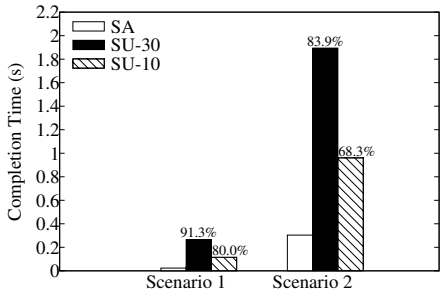
Figure 6 shows the completion time required by SA, SU-30, and SU-10 for one reporting. The completion time increases with the data rate and the number of sensors, as respectively shown in Figures 6(a) and 6(b), because the completion time is proportional to the batch size of each reporting and the number of sensors that report simultaneously. Moreover, the increments of the completion time are more obvious under SU than under SA. This is because the greater the amount of remote data or the number of enabled sensors, the more the time SU spends on issuing operations. The result in Figure 6(c) shows that the completion time under SU is almost the same under different wakeup frequencies. This is as expected because SU reports data according to each sensor's report period, regardless of the system wakeup frequency. For SA, the completion time decreases slightly as the system wakeup frequency increases. This is because SA reports data incidentally when the system wakes up; the more frequently SA reports, the smaller the batch size of one reporting and thus the shorter the completion time. Overall, SA can reduce the completion time by between 66.8% and 85.4%

with respect to SU-10, and by between 74.2% and 94.2% with respect to SU-30. This short completion time is achieved mainly because of SA's simplified access model; that is, the client receives data passively without issuing any requests, except when the semantic information and system state need to be updated.

4.4 Usage Scenarios



(a) Energy consumption



(b) Completion time

Figure 7: Efficacy of SA and SU under two use cases

Figure 7 shows the efficacy of SA and SU under two user scenarios, namely lifelogging and yoga class. The first scenario generates a light data workload but high system wakeup frequency; contrarily, the second scenario generates a heavy data workload but low system wakeup frequency. This explains why the energy consumption is lower and the completion time is shorter under the first scenario than under the second. As observed between SU-30 and SU-10 in both scenarios, SU-10 requires more energy but a shorter completion time, while SU-30 requires less energy but a longer completion time; there is an evident trade-off between the energy consumption and the completion time under SU. However, SA consumes less energy than SU-30 and has a shorter completion time than SU-10 in both scenarios. The results in Figure 7(a) show that SA can reduce the energy required by SU-10 and SU-30 by 9.6% and 32.6%, respectively, under the first scenario; for the second scenario, the energy reductions are 31.9% and 47.9%. On the other hand, the results in Figure 7(b) show that SA can reduce the completion times required by SU-10 and SU-30 by 80% and 91.3%, respectively, under the first scenario; for the second scenario, the time reductions are 68.3% and 83.9%. This result also highlights the efficacy of SA, which aligns reporting times with the mobile system's original wakeups to reduce the additional energy consumed by remote sensing and uses an asynchronous access model to speedup data acquisition. The efficacy is more obvious when the data workload is heavy.

5. CONCLUDING REMARKS

This paper presents a semantics-aware design for mounting remote sensors at the sensor HAL on mobile systems. To demonstrate the efficacy of our design, we implemented it in Android and conducted experiments on LG and Samsung smartphones with popular mobile apps installed. Compared with a semantics-unaware direct extension, our design can reduce the energy consumption by between 9.6 and 47.5%, as well as the completion time by between 66.8 and 94.2%, depending on the data workload and original wakeup frequency. The reduction is due to our asynchronous access model simplifying the inherently synchronous model to save communication energy and time. Moreover, our reporting policy aligns data reporting with the mobile system's wakeup pattern resulting from other apps or system services, to avoid extra wakeup energy costs. Our design remains workable on smartphones with factory settings and is particularly effective for remote sensors that constantly generate a large volume of data. For future research, we will extend our semantics-aware design to jointly consider the energy consumption of not only the system that mounts remote sensors, but also the system that hosts remote sensors.

Acknowledgement

This work was supported in part by the Ministry of Science and Technology under grant 104-2628-E-001-003-MY3.

REFERENCES

[1] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A System Solution for Sharing I/O Between Mobile Systems. In *Proc. of ACM MobiSys*, pages 259–272, 2014.

[2] Asahi Kasei Microdevices Corporation. Sensor Daemon for Android. `https://github.com/akm-multisensor/AK8975_FS/`.

[3] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma. PRISM: Platform for Remote Sensing Using Smartphones. In *Proc. of ACM MobiSys*, pages 63–76, 2010.

[4] Google. The Google Fit SDK. `https://developers.google.com/fit/`.

[5] J. J. S. Hernández. Android SensorList. `https://github.com/josejuansanchez/android-sensors-overview/`.

[6] JekApps. SoundSeeder Music Player. `http://soundseeder.com/`.

[7] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. SeeMon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments. In *Proc. of ACM MobiSys*, pages 267–280, 2008.

[8] M. Keally, G. Zhou, G. Xing, and J. Wu. Remora: Sensing resource sharing among smartphone-based body sensor networks. In *Proc. of IEEE/ACM IWQoS*, pages 1–10, 2013.

[9] Y. Lee, Y. Ju, C. Min, S. Kang, I. Hwang, and J. Song. CoMon: Cooperative Ambience Monitoring Platform with Continuity and Benefit Awareness. In *Proc. of ACM MobiSys*, pages 43–56, 2012.

[10] M. Levin. Designing Multi-Device Experiences: An Ecosystem Approach to User Experiences Across Devices. O'Reilly, 2014.

[11] C.-H. Lin, Y.-M. Chang, P.-C. Hsiu, and Y.-H. Chang. Energy Stealing - An Exploration into Unperceived Activities on Mobile Systems. In *Proc. of IEEE/ACM ISLPED*, pages 261–266, 2015.

[12] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The Jigsaw Continuous Sensing Engine for Mobile Phone Applications. In *Proc. of ACM SenSys*, pages 71–84, 2010.

[13] A. Misra and L. Lim. Optimizing Sensor Data Acquisition for Energy-Efficient Smartphone-Based Continuous Event Processing. In *Proc. of IEEE MDM*, pages 88–97, 2011.

[14] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is Keeping My Phone Awake? Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In *Proc. of ACM MobiSys*, pages 267–280, 2012.

[15] K. Pavel. Anroid IP Webcam. `http://ip-webcam.appspot.com/`.

[16] B. Priyantha, D. Lymberopoulos, and J. Liu. LittleRock: Enabling Energy-Efficient Continuous Sensing on Mobile Phones. *IEEE Pervasive Computing*, 10(2):12–15, 2011.

[17] Samsung. The Remote Sensor SDK. `http://developer.samsung.com/resources/remote-sensor/`.

[18] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. of USENIX ATC*, pages 119–130, 1985.

[19] R. Steckler. Cinch. `http://www.perfectcinch.com/`.

[20] Urbandroid Team. Sleep as Android. `http://sleep.urbandroid.org/`.