Open Data Kit Sensors: A Sensor Integration Framework for Android at the Application-Level

Waylon Brunette, Rita Sodt, Rohit Chaudhri, Mayank Goel,
Michael Falcone, Jaylen VanOrden, Gaetano Borriello
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
{wrb, rsodt, rohitc, mayank, mfalcone, dutchsct, gaetano}@cse.uw.edu

ABSTRACT
Smartphones can now connect to a variety of external sensors over wired and wireless channels. However, ensuring proper device interaction can be burdensome, especially when a single application needs to integrate with a number of sensors using different communication channels and data formats. This paper presents a framework to simplify the interface between a variety of external sensors and consumer Android devices. The framework simplifies both application and driver development with abstractions that separate responsibilities between the user application, sensor framework, and device driver. These abstractions facilitate a componentized framework that allows developers to focus on writing minimal pieces of sensor-specific code enabling an ecosystem of reusable sensor drivers. The paper explores three alternative architectures for application-level drivers to understand trade-offs in performance, device portability, simplicity, and deployment ease. We explore these tradeoffs in the context of four sensing applications designed to support our work in the developing world. They highlight a range of sensor usage models for our application-level driver framework that vary data types, configuration methods, communication channels, and sampling rates to demonstrate the framework's effectiveness.

Categories and Subject Descriptors
D.2.11 Software Architectures

General Terms
Design, Experimentation, Performance

Keywords
Mobile computing, drivers, smartphones, ICTD, sensing, Bluetooth, USB, Open Data Kit.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiSys'12, June 25–29, 2012, Low Wood Bay, Lake District, UK.
Copyright 2012 ACM 978-1-4503-1301-8/12/06...$10.00.

1. INTRODUCTION
Market penetration of smartphones as a computing and communications platform has increased significantly in recent years. Basic feature phones are gradually being replaced by relatively inexpensive smartphones in developing countries. For example, in Kenya the Android based Huawei Ideos is sold for approximately USD 80[27]. Researchers and practitioners in the information and communication technologies for development (ICTD) community are increasingly leveraging smartphones to improve information management in under-resourced environments. Our work is motivated by the platform shift from traditional PCs and standalone sensing appliances to mobile devices (*e.g.*, smartphones, tablets) coupled with cloud services to create mobile information systems. There is an unprecedented opportunity to integrate consumer mobile devices with external sensors enabling the collection of data directly on these devices. However, unlike traditional personal computing devices, the new consumer devices are locked by service providers or manufacturers, and most end-users do not have the administrative rights, technical ability, or organizational capacity to modify or customize the operating system. As a result, relying on conventional in-kernel device driver frameworks to integrate external sensors with consumer smartphones is impractical. Our project explores ways to package software so that non-technical users can access external sensors from a locked mobile device running a stock version of the Android operating system. The framework assumes the consumer device is 'locked down' and an end-user only has the skills to install applications from a standard app marketplace such as Google Play (Google's Android app store).

The ICTD community has begun investigating using phone-based sensing to perform in-situ and remote monitoring [3, 5, 9]. Even though capturing sensor data directly eliminates many of the errors that plague traditional data collection techniques, such as manual form-filling, it is still not widely used in developing regions because of the high level of technical expertise required to develop a mobile sensing application. The technical challenges include managing the details of different physical communication channels, processing sensor-specific data, developing a user interface and designing application control logic. Unfortunately, this level of expertise is usually not readily available in developing regions or even in developed regions on projects undertaken by resource-limited organizations (such as non-profits and community groups). Because of these complications, we hypothesize that including sensors in mobile data collection poses several technical barriers that, if reduced, would enable more applications to leverage sensors for data collection across varied domains. The Open Data Kit (ODK) Sensors framework aims to

lower these barriers by simplifying the deployment of smartphone applications that use external sensors. More specifically, the goal of this work is four-fold:

1. Create a modular framework for adding new sensors by abstracting away management of discovery, communication channels, and data buffers. Integrating a new sensor should require adding only its data handling and configuration primitives.
2. Provide a high-degree of isolation between applications and sensor-specific code. Applications should continue to function even if sensor-specific code is buggy or a sensor becomes inoperative.
3. Understand the tradeoffs of several architectural approaches, especially modularity and performance.
4. Facilitate the integration of new sensors into applications by making it possible to download new sensor capabilities from an application market rather than requiring modifications to the OS configuration.

The ODK Sensors framework provides a single sensing interface for both built-in and external sensors. Having a single interface is appropriate for lightly trained technical workers because it hides a large number of the details involved in developing sensing applications. The framework also provides a simple, high-performing, and flexible abstraction on which to develop and deploy user-level device drivers on Android. While a device driver abstraction is a standard concept, the framework includes features that make development of device drivers easier by handling sensor state (*e.g.,* connection, buffered data, threading) and only requiring driver developers to implement sensor-specific commands and data processing. To evaluate and demonstrate the efficacy of the framework we implemented four applications that are exemplars of different classes of sensor data collection. Three of these applications were previously deployed in developing regions and were ported to the framework leading to significant code simplifications. We discuss these applications and how they leverage the ODK Sensors framework in detail in our previous work [4]. Here we use the applications to demonstrate the benefits of the ODK Sensors framework and compare their minimum performance requirements to the framework's throughput.

This paper examines the architectural implications of three alternative framework architectures that utilize different inter-process communication mechanisms. By comparing peak throughput across the: three framework versions, communication channels, and applications, we show that framework throughput is not the limiting factor of the sensing system. Therefore, our design choices are biased towards making it easier to create mobile sensing applications by focusing on how programming and deployment barriers can be reduced rather than on the relatively small differences in performance.

2. ODK and ODK SENSORS

Open Data Kit (ODK) [11] is a successful suite of mobile tools that exploit the rich interaction and high-performance computing capabilities of smartphones to improve information collection, distribution, and decision-support. ODK focuses on deployment contexts where conventional computing solutions (*i.e.,* informed by concerns of the developed world) are often inappropriate due to constraints such as affordability, infrastructure, institutional capacity, and technical support. ODK Sensors expands ODK by creating a framework to ease the augmentation of a mobile consumer device with sensing capabilities. The ODK Sensors framework supports a variety of external sensors that vary by the type of data they collect, the communication channel over which they interact with the smartphone, and the rate at which they generate data. It provides a unified interface for sensing on Android devices by combining both built-in and external sensors into a single interface. While this design maximizes the variety of sensors available through a uniform interface, the gains in ease of development are more significant for external sensors as more programming is required to interface these as compared to built-in sensors. ODK Sensors focuses on ease-of-use, in general, with a particular focus on appropriateness to our target contexts.

The framework reduces the complexity of building sensor-based mobile applications by providing abstractions that encapsulate communication channels in addition to delineating user-application functionality from sensor communication. The framework has three constituencies: Application Users, Application Developers, and Sensor Driver Developers. A typical Application User is assumed to be the least technically proficient of the three and is only expected to be able to use applications on an Android device. An Application Developer is expected to know how to create new Android applications (design UIs and implement application domain logic), but is not expected to have detailed knowledge of the specifics of sensor control or how the sensors represent and communicate their data. A Sensor Driver Developer is the only constituent expected to understand the low-level protocol used by a specific sensor for configuration and data packaging, but is not expected to deal with communication channel setup or multiplexing. The delineation of application logic from framework logic leads to a clean separation of developer roles and allows an application developer to focus on higher-level application specific concepts while a driver developer focuses on creating sensor-specific drivers.

The goal of the ODK Sensors project is to shift as much responsibility as possible to the framework developers to simplify the creation of sensing application while maintaining a high-level of flexibility for integrating new sensor types. By creating a framework to isolate these three development roles we hope to make it easier to create sensing applications by isolating development tasks that can be fulfilled independently by people with the appropriate levels of technical skill. To encourage new driver development, the framework assumes as much responsibility as possible for aspects common to many sensors, including management of connection state and threads. Additionally, decomposing the system into modules enables more effective testing and code reuse, thereby improving overall system robustness which is particularly important for ICTD deployment settings (since once a system is deployed in remote locations updating it in the field becomes logistically difficult in terms of costs, time, and complexity).

For the framework to successfully enable an ecosystem of external sensors it must be:

1. easy to create sensor drivers, that is, minimizing the knowledge and amount of code required to create a driver,
2. easy to integrate/reuse external sensors in a wide variety of applications,
3. easy to deploy the framework and device drivers, shielding an end user from the technical details of the sensing infrastructure,
4. easy to upgrade the framework and sensor drivers,
5. hard for bad driver code to damage the framework since Sensor Driver Developers may not be expert Android developers,

6. easy for an Application User to discover available sensors through a streamlined user interface, and
7. easy to manage communication channel details such as proper handling of dropped connections.

ODK Sensors attempts to meet all the above requirements by creating an environment of reusable components for the development of mobile sensing applications.

3. RELATED WORK

The variety and number of mobile applications has increased due to the popularity of smartphones and app stores. Despite this proliferation, there are still only a limited number of applications that make use of external sensing devices. This is in part due to the programming challenges of implementing communication between smartphones and external sensors and in part due to resource constraints that prevent adoption of these applications in under-resourced environments. This leads to two main areas of research: (1) reducing programming barriers [8, 21] and (2) making mobile sensing applications more efficient [22, 23, 26, 29]. Within these two areas, some related work focuses on on-device sensors, while other work seeks to expand communication to sensors not built into the phone itself. There is also a significant body of research in device driver design that examines tradeoffs of reliability, ease of use, and performance with user-level versus kernel-level drivers or a combination of the two [10, 17, 19, 20, 24, 25].

The concept of user-level drivers (or application-level) is not new; the L3 system incorporated user-level drivers in 1988 [20]. Leslie *et al.* [19] built user-level device drivers into Linux without significant performance degradation, even for high-bandwidth devices such as Ethernet, by implementing a framework that used shared data structures, batched work, and optimized event notification. Microdrivers [10] developed a program to split existing drivers into kernel-level and user-level parts by leaving critical path code in the kernel (*e.g.* data handling, I/O) and moving the rest of the driver code to a user-mode process. Similarly, Decaf Drivers [24] implemented ways to convert Linux kernel drivers to Java programs running in user mode. These systems demonstrated good performance, despite not using native kernel drivers. While ODK Sensors was influenced by these projects, it focuses on creating user-level drivers for locked consumer devices running Android. Therefore, unlike these projects we do not alter the kernel to provide the communication link between the OS and the user-level driver. Instead, ODK Sensors' communication managers run as user-level threads and use Android's APIs to handle sending and receiving data from the sensor and then forward the bytes to the appropriate device driver for processing.

The migration towards user-level drivers is in part motivated by the desire to make systems more fault tolerant and reliable in the face of driver-error. Maverick [25], a web-based system, provides security by using device drivers and frameworks that run as user-level web applications to support interacting with multiple USB devices. Alternatively, Carburizer [16] detects and tolerates interrupt-related bugs to proactively manage device failures for improved reliability in the presence of faulty devices. Like Maverick, ODK Sensors leverages user-level drivers to provide reliability and security; however, ODK Sensors runs each driver as a separate application causing each driver to be isolated in its own virtual machine.

Other frameworks similar to ODK Sensors have been proposed, but they seek to interface primarily with built-in sensors. Zhuang *et al.* [29] introduced an adaptive location-sensing framework that improves the energy efficiency of location-based applications through suppression or substitution of location requests from built-in GPS sensors. It seeks to increase energy efficiency of the system, which differs from our goal of lowering programming barriers.

Dandelion [21] supports building applications distributed across a Maemo Linux smartphone and wireless body sensors by providing abstractions that shield application developers from hardware specific code. Dandelion envisions a scenario where sensor vendors provide a runtime to enable a platform-agnostic programming abstraction called a 'senselet' written by application developers to run on the sensor itself. The ODK Sensors framework also shields application developers from sensor-specific hardware; however, the framework provides abstractions at a different level as sensor drivers execute on the smartphone and leverage the framework's communication channel abstractions and sensor state management. The initial processing of sensor data occurs on the Android device in the sensor driver (removing this concern from the scope of application developers), whereas Dandelion requires data processing in the 'senselet' on the sensor that must be written by the application developer in this limited sensor environment. Additionally, ODK Sensors does not require sensor vendors to include a runtime enabling the framework to support any standard sensor that communicates via a supported communication channel. The Reflex [22] project (a fork of Dandelion) is a suite of runtime and compilation techniques that conceals the heterogeneous distributed nature of the system and reduces power consumption by offloading data processing to lower-power co-processors. While Reflex focuses on energy efficiency and performance in mobile-sensing applications, ODK Sensors focuses on lowering programming barriers for application developers and supporting different data and application types. The ODK Sensors driver executes within the framework rather than on a separate co-processor. LittleRock [23] and Turducken [26] have similar goals as Reflex, and present other architectures that offload continuous sensor data processing to dedicated low-power processors.

Gadgeteer [28] is a rapid prototyping platform that eases development with embedded hardware devices through the use of modular hardware components and object-oriented programming in C#. While Gadgeteer and ODK Sensors are both focused on making it easier for users to integrate with different external sensors, Gadgeteer achieves this by simplifying how different hardware pieces talk to each other, whereas ODK Sensors aims to make it easy for the mobile application developer to leverage a variety of sensors in their application without significant programming knowledge about the specific sensor.

IOIO [15] is a development board designed to work with Android phones through a USB connection. It abstracts the communication between external hardware and software running on the smartphone, enabling Android applications to directly control hardware attached to the IOIO board. It is different from ODK Sensors because IOIO provides an abstraction to Android applications at the level of I/O pins of the IOIO board. ODK Sensors currently interfaces with Arduino boards to enable low-power sensing, by decoupling the interface board from the Android application. An independently operating sensor board enables sensing to occur at lower power allowing the Android device to remain in a sleep state longer. Amarino [18] is another toolkit that connects Android phones with Arduino microcontrollers via Bluetooth. It helps developers in easily sending information about phone's internal events such as phone calls, SMSs, and on-device sensor data over Bluetooth. It is

similar to the IOIO Board but with communication over Bluetooth rather than USB.

AndWellness [12] lets researchers customize surveys to collect data from sensors on phones carried by study participants. It shares our framework's goal of lowering barriers for building sensing applications. However, unlike ODK Sensors, this application focuses primarily on customizing surveys and front-end visualizations of real-time data. Our work aims to lower barriers for the application and sensor driver developer and in the future interface with applications such as ODK Collect [11] or ODK Tables [13], which will help make it easier to develop information services directly by end-users.

PRISM [8], like ODK Sensors, is sensing application middleware whose aim is to provide reusable components and eliminate redundant efforts regarding distributed operation, security and privacy. PRISM has been evaluated for a variety of applications, all of which interface exclusively with sensors built into the phone. PRISM also has a focus on deploying these applications at scale. In contrast, our framework focuses on interaction with external sensors by abstracting away the communication layer to make programming easier.

These related projects address many problems that are common to mobile sensing applications. We aim to further lower development barriers by simplifying the process of connecting a smartphone to an external sensor through the creation of a framework with tailored abstractions that facilitate the integration of new and varied types of sensor data into mobile applications.

4. FRAMEWORK

The ODK Sensors framework simplifies the development of sensor-based mobile applications by creating a common abstraction point that enables all sensors to be accessed through a unified interface. Creating a single-interface reduces complexity since all external sensors as well as Android's built-in sensors are exposed through a common interface regardless of the communication medium used. The interface encapsulates communication and delineates user-application code from sensor-specific driver code, freeing application developers from understanding the specifics of the underlying communication between an Android device and an external sensor. From a user's perspective the overall architecture for ODK Sensors consists of three apps: the *User-Application App,* the *ODK Sensors Framework App,* and the *Sensor Driver Apps*. For the purposes of this paper an Android application (software downloaded from a market) is referred to as an "app", whereas the word "application" is used to refer to usage/deployment examples. The *ODK Sensors Framework App* is responsible for managing low-level, channel-specific communications and providing abstractions to isolate sensor driver code. The *User-Application App* communicates with sensors through the unifying framework API (explained in Section 4.1). Figure 1 shows an end-user view of ODK Sensors on a smartphone. In the figure, two apps (*User-Application App* and *ODK Sensors Framework App*), a *USB Bridge*, and a temperature probe are used to monitor the flash heat pasteurization of milk to eliminate contaminants (*e.g.* HIV) in breast milk (described in section 5).

We chose Android as the target platform for the ODK Sensors framework because it is open source and has extensive support for background processes and includes several built-in constructs for inter-application communication (IPC) between Android Applications. Examples these Android constructs are detailed in Table 1. For instance, a *Broadcast Receiver* uses non-blocking message passing to communicate between applications; whereas, a *Service* construct communicates through a blocking inter-process communication mechanism. The ODK Sensors framework uses the Android Interface Definition Language (AIDL) to specify the programming interface that both the client and service use to communicate. From the AIDL, Android generates IPC code to decompose objects into primitives that the operating system can marshal as *parcels* across process boundaries to provide blocking IPC functionality. These constructs enable a comparison between a single-threaded asynchronous model (*Broadcast Receiver*) and a multi-threaded synchronous model (*Service*) for inter-application communication. Additionally, Android supports multiple communication APIs that facilitate connecting to a wide variety of external sensors. While APIs for Bluetooth and Wi-Fi radios have long been available on Android devices, only recently has Android added USB support through the Android Accessory Protocol (AAP) [1] enabling compliant devices to connect to external hardware over USB. The AAP requires the external device to act as the USB Master while the Android device is the Slave. To include a wide variety of external sensors in the framework, we utilize a *USB Bridge* to connect sensors that use diverse digital I/O protocols (*e.g.* I^2C, SPI) to the phone's USB port via an interface board. For our initial prototype, we used an Arduino board [2] to act as the *USB Bridge*.

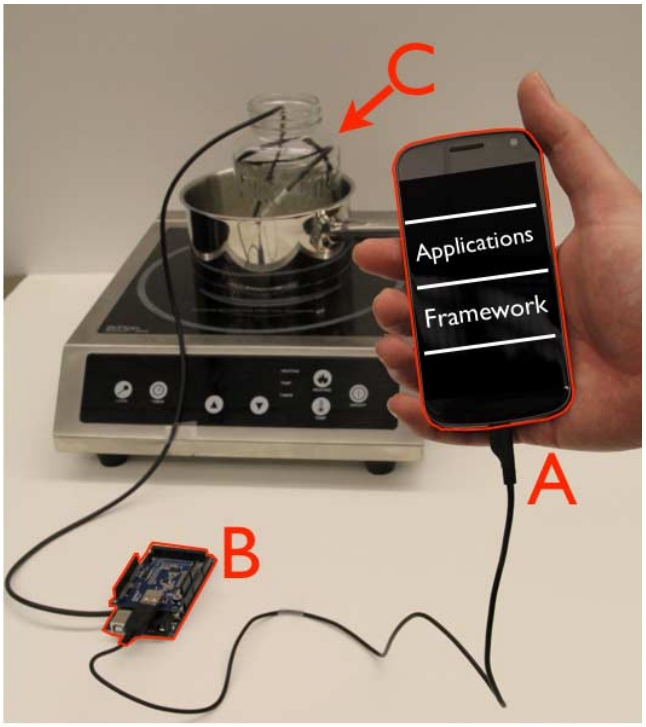

Figure 1: End-user view of a smartphone using the ODK Sensors framework to connect to a temperature sensor via an Arduino *USB Bridge* for the milk pasteurization application (Section 5 describes the application). The *Application* and *Framework* are Android apps that are installed on the mobile device (A). The mobile device is connected to an Arduino interfacing board (B) over USB; forming a *USB Bridge* to the temperature sensor (C) using the Arduino board's I/O ports (B). The *Application* uses the *Framework* to get data from the temperature sensor over USB using the ODK Sensors API.

Table 1: List of Android constructs used in the three framework implementations. The constructs listed are used for application metadata, packaging, storing, or sending data between applications and forming the basis of inter-process communication with sensor drivers in V2 and V3 (Section 4.2)

Android Construct	Description
Service	Services run in main thread of hosting process. Requests from other processes are handled concurrently by a threadpool.
Broadcast Receiver	Asynchronous broadcast receiver. Receives broadcasts sent via Intents from other processes.
Intent	Asynchronous messages that represent an operation to be performed, such as broadcasts, start services, start application, *etc*.
Bundle	Mapping from String values to parcel-able types. A *parcel* is a data container used for IPC.
Manifest	Provides essential information about the application (permissions, package name, etc.) to the Android OS.
Content Provider	Stores and retrieves data that is accessible by all applications.

The ODK Sensors framework (shown in Figure 2) presents a common interface to all top-level user applications via the Service Interface and Content Provider. User-Application apps only need to implement the application-specific logic that handles processed sensor data received from the framework. For each call to the service, the *Sensor Manager* dispatches the commands to the appropriate sensor object that, in turn, utilizes a sensor driver to perform specific low-level tasks. The framework supports multiple communication modalities by providing abstractions called *Channel Managers* that encapsulate complexities specific to each communication channel. ODK Sensors supports multiple data types, sample sizes, sampling frequencies, and sensor configurations by utilizing *Sensor Driver* abstractions that encapsulate sensor-specific data processing. These abstractions enable applications to interface with sensors using higher-level key-value pair constructs that are not constrained to be fixed-size arrays or values of a specific type. This enables developers to focus on the application logic instead of sensor-specific logic.

The framework's communication subsystems provide abstractions for lower-level, channel-specific communication protocols that make it easier for a driver developer to interface with an external sensor. The framework encapsulates communication channel specifics within the respective channel managers to hide them from application and sensor driver developers. For instance, Bluetooth-enabled sensors need to be discovered and paired with the smartphone and a socket needs to be set up for communication. ODK Sensors automatically manages this entire process for the application developer. The current implementation supports communications over Bluetooth and USB; in the future, we plan to add additional channel managers to support other communication methods such as NFC and Wi-Fi. To create a single unified sensing interface, the ODK Sensors framework also exposes all 11 built-in Android sensors (for Android 2.3 and greater) creating a single integration point for sensing.

A *Sensor Driver* handles the particular messaging protocol that configures and/or requests data from an external sensor by issuing commands to the appropriate *Channel Manager*. During data collection, the *Communication Manager* passes all raw data received from the sensor to the appropriate sensor driver via the *Sensor Manager*. Sensor drivers receive and process data encoded in formats specific to their respective sensors and generate configuration commands as required by the sensor. The sensor driver parses this sensor-specific data, transforming it into key-value pairs that can be easily consumed by top-level user applications. Likewise, the sensor's configuration parameters are specified as key-value pairs by user applications and passed to the appropriate *Sensor Driver* by the *Sensor Manager*. The driver encodes these key-value pairs according to the sensor's messaging protocol and the encoded data is sent to the sensor via the *Channel Manager*. By having channel managers handle communication details, the driver developer no longer needs to be aware of channel-specific protocols. Instead they can simply implement the interface described in Section 4.2 and convert raw sensor data coming from the communication channel into key-value pairs and vice versa for sensor configuration data.

In addition to allowing for easy reuse, the *Sensor Driver* design shields applications from changes in the communication protocol, configuration, or data type. Shielding applications from driver or channel changes leads to more robust systems that are easier to maintain. The sensor driver abstraction also enables multiple apps to interface with the same type of sensor by reusing an existing *Sensor Driver*. The framework's sensor drivers are designed as stateless processors of data to shield driver developers from tasks required to enable interaction with multiple identical sensors simultaneously (*e.g.,* channel management, threads, buffers).

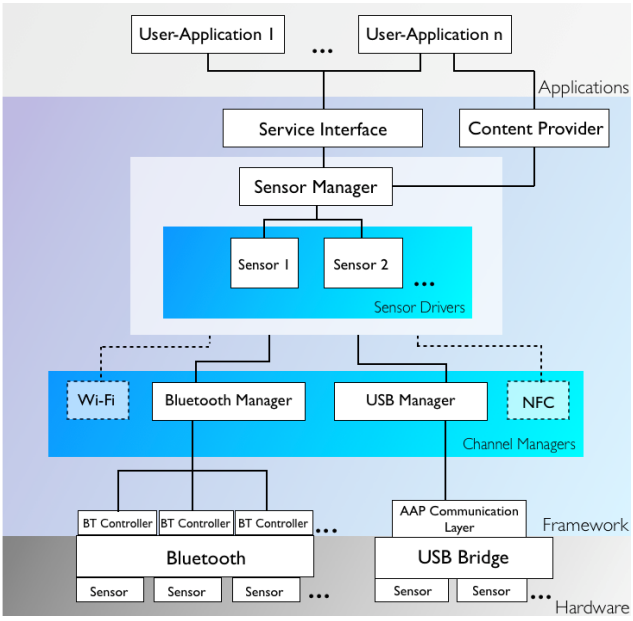

Figure 2: Architecture overview of ODK Sensors system. Sensor Manager maintains references to all sensors and the corresponding sensor drivers. Channel Managers manage the connections over the communication channels (e.g., USB, Bluetooth). The Service Interface and Content Provider provide the access point for applications to interact with the framework. Note: All architectures use the same data flows; however, in V2 and V3 sensor drivers move to separate apps as described in Section 4.2.

To better understand which Android IPC constructs to use to create our application-level (or user-level) sensor driver framework we implemented three different versions of the framework. The *Sensor Manager* and *Channel Managers* communicate with sensor drivers either with method calls inside the framework (V1), with IPCs through a Service Interface (V2), or with broadcasts to Broadcast Receivers (V3). To establish a baseline to compare against, version 'V1' of the framework was created as a single app. The device drivers in the framework are stateless processors of data, thus allowing the driver programmer to keep code simple by avoiding managing multiple sensors within the driver or having to run multiple apps for each sensor type. The details of the interfaces presented to developers and framework's communication subsystems are discussed in the following subsections.

4.1 Framework Interface

The ODK Sensors framework interface abstracts many sensor specific details and does not expose any channel specifics. The *Service Interface* creates a common interface for user applications to leverage both built-in sensors and external sensors connected over the communication channels through methods shown in the following code snippet:

```
interface ODKSensorService {
  boolean sensorConnect(in String id, boolean
      useContentProvider);
  void configure(in String id, in Bundle config);
  boolean startSensor(in String id);
  boolean stopSensor(in String id);
  List<Bundle> getSensorData(in String id, long
      maxNumReadings);
  boolean isConnected(in String id);
  boolean isBusy(in String id);
  boolean hasSensor(in String id);
}
```

The *Service Interface* requires the sensor's ID to control a specific sensor in the framework. If applications do not know the Sensor ID, then the framework provides an interactive discovery process for users, freeing the application from implementing their own sensor discovery interface. To launch the ODK Sensors interactive sensor discovery UI, the application simply sends an Intent to the framework as described in Section 4.2.4. Once the application has the sensor ID, it can connect to the sensor and begin retrieving sensor data by periodically calling the `getSensorData` method of the Service Interface. This method returns sensor readings to the application in a mapping of key-value pairs created by the *Sensor Driver* processing raw sensor data. As an example, a temperature sensor's driver parses data according to the messaging protocol of the sensor to extract multiple (or single) temperature readings in degrees Celsius and passes a list of Android *bundles* that map key = "temperature" to value = "value in °C" to the application. The framework expects the application to retrieve data in a timely fashion to clear the memory that is buffering the data. In cases where the application does not plan to read the data immediately, the framework provides applications a second mechanism for retrieving sensor data stored in a local database through a *Content Provider*. Applications specify whether the framework should put the data in a database to be accessed by a Content Provider when calling `sensorConnect`. An application can query the Content Provider whenever it wants to get data from a sensor that the framework considers owned by the application.

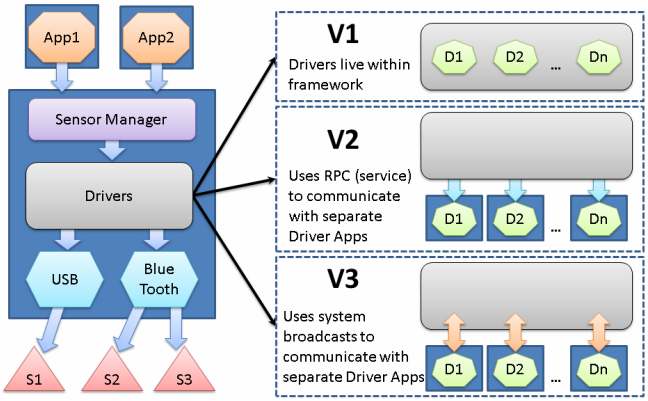

Figure 3: The three different framework implementations that vary with respect to how the framework communicates with the sensor drivers. Each dark rectangle represents a separate application. The design in which the drivers live within the framework is referred to as V1. The V2 design uses remote service calls to communicate with separate driver apps outside the framework. V3 uses system broadcasts to communicate with these separate driver apps.

4.2 Sensor Drivers

Sensor drivers are designed to abstract sensor specific control code from more general sensor management code. Certain concepts are common to all sensors, such as initialization, configuration, and taking readings, but the framework does not need to, nor should it know, specifically, how each sensor accomplishes these tasks. Sensor drivers enable the framework to communicate with sensors while maintaining the necessary abstractions that keep the framework modular and extensible. The same driver interface is used for all sensors – both built-in and external sensors – as the driver interface abstraction encapsulates sensor-specific data transfer and processing, and hides sensor specifics such as data types, frequency of collection, data size, and various configuration parameters. The driver abstractions enable the framework to reuse core functionality such as sensor configuration, connection, communication handshakes, buffering data, multiplexing, etc. for multiple types of applications. Simultaneous integration of different sensors involves complex tasks such as concurrent Bluetooth and USB setup that requires multiple data sockets and threads to buffer and process the data from these connections. The framework hides these complexities thereby significantly simplifying the job of both the application-level developer and the sensor driver developer.

Moving sensor drivers into separate Android apps that implement a common driver interface improves the framework modularity and extensibility. Separating drivers from the framework enables drivers for new sensors to be downloaded and installed onto an Android device from any Android marketplace or website like any other Android app. To understand the various architectural trade-offs of moving sensor drivers to external Android apps, three different framework architectures were created to compare two different Android IPC mechanisms. As depicted in Figure 3, one design keeps sensor drivers within the framework (V1) serving as a baseline and two designs move each driver to its own external Android app (V2 & V3). To understand the best way to communicate with the external drivers one framework used a blocking Binder IPC call (V2) while the other used a non-blocking message passing structure (V3). Both of these versions enable end-users to dynamically add drivers to the ODK Sensors

framework as needed. User applications remained the same across the three implementations, as their interface with the framework did not change. In each of the three framework implementations every driver must implement the following interface:

```
public interface Driver {
  byte[] configureCmd(Bundle config);
  byte[] getSensorDataCmd();
  SensorDataParseResponse getSensorData(long
       maxNumReadings, List<SensorDataPacket>
       rawSensorData, byte [] remainingData);
  byte[] startCmd();
  byte[] stopCmd();
}
```

Drivers implement this interface to specify how to parse raw sensor data and what, if any, sensor-specific messages to send for configuration, getting data, and starting or stopping the sensor. The device drivers in the framework are stateless processors of data, which means the programmer does not need to manage multiple sensor instances. Methods such as `getSensorData` require the sensor driver to maintain access to buffered data it has previously processed as well as the raw data from the sensor it has not yet processed. To facilitate this `getSensorData` returns a `SensorDataParseResponse` which is simply a list of key-value pairs that have been fully parsed as well as any leftover bytes from the input stream. We eliminate state from the driver itself and instead have the framework handle all sensor state, including buffered data, and provide it to the driver at the appropriate time when the application has requested data. Implementing an external sensor driver for each of the three architectures is slightly different as described in the following subsections. In the case of the built-in sensors, their drivers are included with the framework even though it is possible to implement them as separate apps since communication is handled internally by the Android OS. The commands are Android-specific, not communication channel specific so it seems unnecessary to move them outside of the framework.

4.2.1 V1: Drivers within the Framework

In the first version, V1, we place all the sensor drivers inside the framework thus creating a single Android app. At runtime, the framework accesses individual sensor drivers from an in-memory map. In this architecture, the drivers, by executing within the framework, create a tight coupling that should have the best performance and provide a baseline against which to compare the other versions. However, this design is not ideal for our target users to dynamically add new drivers as it requires a recompilation or the use of a class loader. Using a class loader is not ideal for a population of non-technical users because it requires placing files into proper access-controlled directories on the device, thereby enabling the framework to dynamically load classes. Our goal is to build a system that uses established Android distribution and communication mechanisms to dynamically deliver and add drivers to the framework.

4.2.2 V2: Driver Communication via Services

The V2 version of the framework creates separate Android apps for each sensor driver. Each of these Apps implements an Android *Service* that defines the `Driver` Interface using AIDL, enabling the framework to communicate with the sensor driver via Android's Binder IPC. Android provides an AIDL to define the programming interface between the client and server and automatically generates stub Binder IPC classes. When the client calls these methods, the system copies the payload from the client into kernel memory, which is memory-mapped into the server's address space. The server-side procedure then handles the call. The sensor driver application includes driver-specific metadata that is used by the framework for driver discovery (described in Section 4.2.4). If an appropriate driver application is installed on the device, a generic sensor object is constructed to act as a proxy between the driver and framework by binding to the `Driver` interface that is presented as an Android Service. The framework maintains a reference to the specified generic sensor object that communicates with the driver using Android's Binder IPC. Each driver proxy acts as a thin wrapper for the driver and contains a reference to the appropriate channel manager. Communication with the sensor forwards commands returned from the driver so that they are transmitted over the appropriate channel. Since the driver application can be reused by multiple instances of the same type of sensor it is important that it be stateless. Therefore, the driver's remote function calls are designed to transfer the required state to the driver on each service call and allow the driver to return state information along with the parsed data. An example of state information stored by the framework is the excess bytes from the input data stream, enabling buffering of the input stream until enough data is received to parse and produce a full message.

4.2.3 V3: Driver Communication via Broadcasts

The third and final version of the framework, V3, also implements sensor drivers as separate Android applications but uses message passing with *Broadcast Receivers* for IPC (instead of synchronous blocking IPC). Each driver specifies a unique broadcast address that is discovered by the framework during the driver discovery phase (described in Section 4.2.4). The framework communicates with the driver by sending messages to the driver's unique broadcast address. Included in the broadcast message sent to the driver is a unique broadcast address for the driver to use to send a response back to the framework. The additional information included in each of the *Intents* (*i.e.* the data exchanged) is part of the API between the framework and drivers. By instantiating a broadcast receiver for each instance of a sensor, the framework can multiplex responses for each individual sensor. This API provides the same functionality as the interface implemented by sensor drivers in V1 and V2. Similarly as for V2, the driver does not maintain state, allowing the driver application to be reused by multiple instances of the same sensor type. The message passing interface is designed to communicate state between the framework and the sensor driver, enabling the sensor driver to cache incomplete information between processing sensor readings.

This architecture decouples the drivers from the thread of control allowing for a non-blocking message passing framework but incurs the overhead of using Android *Broadcasts* for data communication. The framework is better shielded from buggy drivers due to the decoupling enabled by the blocking semantics. However, we acknowledge that broadcasts can be intercepted by another Android application, which is a security risk. This will be addressed in future work by encrypting data sent between different processes.

4.2.4 Sensor & Driver Discovery

Each version of the framework semi-automates the sensor discovery process by providing a pre-built UI that application developers can launch when they need the user to select one of the currently available sensors. When the framework gets a request for an unknown sensor, it informs the user's application to launch the framework's built-in sensor discovery system to find and pair the appropriate device driver for the desired sensor. The user simply needs to select the sensor along with the corresponding

driver from the list presented. The framework displays a list of available drivers for the relevant communication channel, as well as a list of sensors that are already physically connected to or in Bluetooth wireless range of the device. In V2 and V3, the framework automatically discovers installed driver applications by searching Android *manifest* files. Once the user has mapped the sensor with the appropriate sensor driver the framework will do the rest. After the user selects the sensor they want to use, the ID is returned to the application. This ID enables the application to control the sensor with the methods presented in the ODK Sensors Service Interface. Once a sensor has been discovered it is stored in the *Sensor Manager* database enabling the application to skip the discovery step in the future.

An advantage of having sensor drivers as separate applications is that it simplifies driver distribution. Ideally, we envision a scenario where manufacturers post their drivers on their own website or on an Android market (such as Google Play). By taking advantage of Google's standard application distribution system the framework is designed to lower barriers for novice users by using familiar application delivery channels. Additionally, if the device manufacturer needs to update their driver they simply need to post the updated application to market and the built-in Android application update system will take care of the rest. Another advantage of separating the device drivers into separate Android apps is it gives manufacturers who want to keep their protocols proprietary an opportunity to create drivers that can be easily used to create applications while protecting their protocols.

5. APPLICATIONS

We developed a few example applications to demonstrate reuse, flexibility, and extensibility of the framework. These applications exemplify the three basic dimensions of variation in sensing applications: *communication channel*, *sensor configuration,* and *data collection style*; all of which must be supported flexibly by the framework. First, the channel used to communicate with the phone can vary across sensors and applications (*e.g.,* USB, Bluetooth, NFC). Second, sensors have different configuration requirements, which may include various parameters or settings that need to be specified such as sampling rate, trigger conditions or alerts, identifiers, and calibration. Third, the data needed by an application can change in format, size, and frequency of collection. The framework aims to support any combination of communication, configuration, and data type transfer between phone and sensor. A top-level user application retains the same interaction with a sensor driver in the framework even if there are changes in communication protocol, configuration, or data type. In the event of such changes, the sensor driver only requires minimal adjustments for parsing a new type of data or specifying a new channel manager.

Developing these applications for the ODK Sensors framework helped to evaluate whether we could reuse the provided abstractions for different use cases, verifying the framework and sensor driver's interfaces. The applications exercise both the wired and wireless subsystems of the framework and, in our experience, are exemplary of the four commonly used modes of data collection in sensor-based systems (Table 2):

- Single Reading: The user requests data from the sensor and chooses to record data points by taking a single reading from a real time stream of data. A clinician's application that connects tools (*e.g.,* blood pressure, pulse oxymetry) to a phone is an example of this use case.
- Real-Time Time-Series: The user of a data collection node has an active session with the sensor and observes a stream of samples from the sensor in real-time. Monitoring the temperature curve of the milk pasteurization procedure [7] is an example of this use case. The temperature curve is also saved so that it can be reviewed later.
- Snapshot Time-Series: Sensors are deployed to autonomously monitor certain phenomena. They aggregate readings internally over a period of time and may report some to a remote location periodically (e.g., alert to detect a specific condition). Temperature and electrical current sensors deployed to monitor vaccine refrigerators are examples of this use case [5].
- Historical Time-Series: Sensors are deployed to autonomously monitor certain phenomena (*e.g.,* movement of an object such as a water can over a period of months). However, unlike a Snapshot Time-Series, data retrieval is not automatic and requires someone to be within range of the sensor to offload the sensor's stored data. The WaterTime monitoring [6] application exemplifies this approach and is dependent on field calibration for configuring the sensor with sampling rates and identifiers.

Table 2: Variations in the sample sensing applications.

Application	Comm. Channel	Configuration	Data Style
Medical	Bluetooth	Calibrate	Single Reading
MilkBank	USB	Sampling Rate	Real-Time Time-Series
Vaccine	USB	Alerts Sampling Rate Snapshot Size	Snapshot Time-Series
WaterTime	Bluetooth	Identifier Calibrate	Historical Time-Series

Three of these applications have already been deployed in developing regions as part of pilot studies. Rewriting these applications to leverage the ODK Sensors framework has simplified them significantly by separating out the application specific logic from the communication logic. Figure 4 shows how these applications interact within the V1 framework.

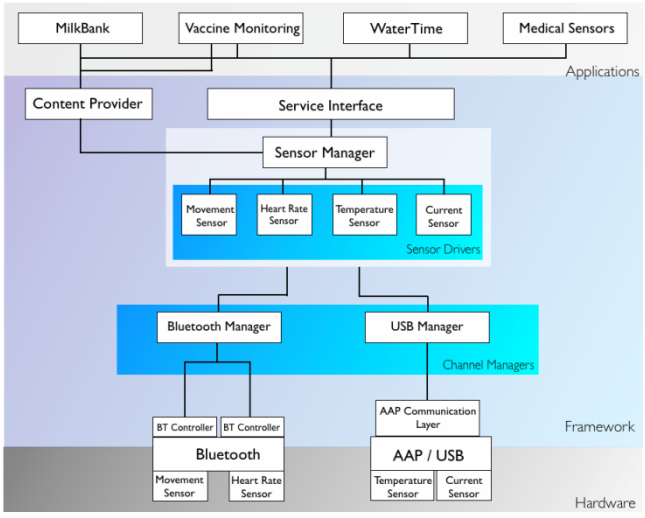

Figure 4: Example instance of V1 framework architecture incorporating the four applications discussed.

6. EXPERIMENTS

The experiments were designed to explore the tradeoffs between the three frameworks in terms of performance, power consumption, and ease of programming. Each of the architectures was benchmarked based on the same set of applications, sensor drivers, and communication channels. The code of the framework and drivers is identical for V1, V2, and V3 with the exception of the protocol the framework uses to communicate with sensor drivers. The protocol changes required four Java files to be different. User applications were not affected across the three implementations, as their interface with the framework did not change.

6.1 Performance

Performance experiments were conducted using several different Android devices to understand the impact of hardware variability. The Samsung Galaxy Tab and Samsung Nexus S were used to represent high-tier Android devices, the HTC Nexus One and Motorola Droid were used to represent mid-tier devices, and the Huawei IDEOS was used to represent low-tier Android devices likely to be common in developing regions. An Arduino Mega2560 with USB Host capability connected to a Motorola Xoom tablet was used to characterize the *USB Bridge*.

6.1.1 Sensor Application Throughput

First we measured the throughput of the four real-world applications described in Section 5. Each test consisted of 60 seconds of continuous data collection. The results in Table 3 show that the rate at which a sensor application gets sensor data is often on the order of one sample every 1 to 2 seconds. In fact, three out of four of the applications collect a data sample no more than twice per second; this rate is actually limited by the sensor itself. The Vaccine and MilkBank applications use a digital temperature sensor that requires a 750ms delay in sampling the ADC, while the heart rate sensor used in the Medical application transmits a packet every 2 seconds. The framework throughput is substantially higher than the saturation point for these three applications. However, in the WaterTime application, which requires a one-time bulk reading of historical data, the sensor sends its collected data as fast as possible over Bluetooth and achieves a throughput of 51 packets per second. This is close to the maximum throughput achieved on the Bluetooth channel using the framework (see Table 4).

Table 3: Observed data throughput from sensor to application on four real-world applications. Packet size is not consistent across applications as the size of data samples varies.

Application	Throughput (pkts/sec)
WaterTime (Bluetooth)	51.0
Medical (Bluetooth)	1.5
Vaccine (USB)	1.0
MilkBank (USB)	1.0

6.1.2 Communication Channel Throughput

Next, we systematically tested the saturation point of the Bluetooth and USB communication channels with a stress test that sent fake data as fast as possible over the channels. To do this we programmed a "spammer" sensor on two Arduino microprocessors; one that emulates a Bluetooth-based sensor and another that emulates a USB-based sensor. Each "spammer" sensor reacts to a "start" signal by executing a loop to send 1 byte data packets as fast as possible until it receives a "stop" signal. In our tests we allowed the "spammer" sensor to send a rapid, constant stream of data for ten minutes on each of our three framework architectures on Bluetooth and on USB. The maximum throughput that could be achieved with full saturation of the communication channels is shown in Table 4.

Table 4: Throughput (pkts/sec) on Bluetooth & USB Channels

Framework	Bluetooth	USB
Drivers in Framework	58.0	42.5
Drivers w/ Service	58.7	41.5
Drivers w/ Broadcasts	49.0	42.0

The throughput results were surprising as the USB channel had lower throughput than the Bluetooth channel. Investigating this issue further revealed that the USB Host driver on the Arduino adds some delays that impact the channel performance. We found that increasing the payload size did not significantly reduce the packets per second rate. After increasing the message size to 1KB per packet and disabling our reliability system, the *USB Bridge* gave a channel bandwidth of over 40KB per second. We found that increasing the payload size beyond 1KB was problematic and was primarily limited by the Arduino's memory size. This is acceptable for now because our real-world USB applications require throughput that is significantly lower than what we can already achieve with our current *USB Bridge*. However, in the future, we will explore alternative options for the *USB Bridge* to achieve higher performance to support more demanding sensing applications, such as those streaming high-resolution camera data.

6.1.3 Framework Throughput

Finally, we evaluated the performance of each framework by establishing an emulated communication channel for fake sensors to use with varying send rates and packet sizes. These experiments tested the throughput of the three framework versions by eliminating the limits imposed by real communication channels. For a baseline understanding, we evaluated the framework's throughput by varying packet size and the delay between packets (Table 5). To understand the effects of multiple sensors, we ran tests that varied the number of sensors that communicated simultaneously through the framework (Table 6). Finally, we verified that the framework's performance did not significantly degrade when operating on different classes of Android devices (high-tier, mid-tier and low-tier) (Table 7). This is important because users in developing regions will likely have a diverse set of devices.

We measured framework throughput (packets/second) by sending data at varying rates with varying sizes as shown in Table 5. The test results reported the number of packets received on average per second for each of the frameworks V1, V2, and V3 on a Nexus One. Each test ran for three minutes on five different Nexus One phones. The results of the tests were averaged. The send delay values began at 1ms and were increased by doubling the delay to a max of 128ms, while the packet size started at 1 byte and was increased by an order of magnitude for each test up to a maximum of 100,000 bytes. We varied these parameters to understand any limiting factors inherent to the different IPC mechanisms. Differences in performance of the various architectures become negligible as the send delay becomes the dominant limiting factor. None of the three frameworks were able to complete the most strenuous tests of 100,000 byte packets being sent with only a 1ms delay because of memory errors invalidating the results (indicated by "Error" in the table). Generally speaking, the throughput values of the framework are similar since the cycles spent in IPC are a small part of the total framework execution time. In the cases where the send delay was not the dominant factor, V1 appeared to perform the best since it

Table 5: Throughput results (pkts/sec) for a Nexus One when varying the packet size and inter-packet generation delay for the three framework versions. The top section of the table contains V1 results, the middle section contains V2 results, and the bottom section contains V3 results. The 'Error' value indicates a test run was unable to be completed because of a memory error. The Max column contains the theoretical maximum throughput.

V1	(bytes) 1	10	100	1K	10K	100K	Max
(ms) 1	798.1	796.9	790.2	747.9	Error	Error	1000.0
2	441.0	440.8	438.9	423.6	Error	Error	500.0
4	234.6	234.5	234.8	229.6	Error	Error	250.0
8	121.4	121.4	121.3	119.7	111.5	Error	125.0
16	61.6	61.6	61.5	60.9	58.1	Error	62.5
32	31.0	31.0	31.0	30.9	30.1	Error	31.3
64	15.5	15.5	15.5	15.5	15.3	13.8	15.6
128	7.8	7.8	7.8	7.8	7.7	7.3	7.8
V2	(bytes) 1	10	100	1K	10K	100K	Max
(ms) 1	785.5	786.8	781.1	Error	Error	Error	1000.0
2	437.6	438.0	436.0	421.8	Error	Error	500.0
4	233.7	233.8	233.0	229.0	Error	Error	250.0
8	120.6	120.6	120.3	119.6	Error	Error	125.0
16	61.4	61.4	61.3	61.0	58.8	Error	62.5
32	31.0	31.0	31.0	30.9	30.1	Error	31.3
64	15.5	15.5	15.5	15.5	15.3	Error	15.6
128	7.8	7.8	7.8	7.8	7.7	7.3	7.8
V3	(bytes) 1	10	100	1K	10K	100K	Max
(ms) 1	771.5	768.0	760.5	Error	Error	Error	1000.0
2	432.2	431.5	426.5	419.2	Error	Error	500.0
4	231.9	231.9	231.2	225.2	Error	Error	250.0
8	119.2	119.1	119.2	118.4	Error	Error	125.0
16	60.9	60.8	60.8	60.4	Error	Error	62.5
32	30.8	30.9	30.8	30.7	30.1	Error	31.3
64	15.5	15.5	15.5	15.4	15.3	Error	15.6
128	7.7	7.7	7.7	7.7	7.7	Error	7.8

does not incur any IPC overhead; whereas, V3 had lower performance since it communicates with drivers uses message-passing.

The primary difference between V2 and V3 is the IPC method. Our results confirm our expectation that the broadcast intent system has more overhead than a synchronous Binder IPC call. This finding correlates with other research that showed the Android system implements the *Intent* call as two IPC-style calls – one from the sender to the system and one from the system to the receiver [14]. While the 'double call' for V3 causes slower performance, the broadcast communication is still only a small amount of the overall work. Therefore, while the effects are visible, they are generally negligible except at faster data generation rates. Our results also matched the findings that there is a drop-off in performance for large packets. The Android system allocates a default kernel-side buffer 4kB in size for each process that will receive data from Binder IPC calls. When the payload size exceeds 4KB, the system allocates an additional, temporary buffer to transfer the data. These larger IPC calls incur additional allocation overhead and were shown to be inefficient as the Android system memory-maps each additional page separately, instead of mapping them all at once [14]. These factors result in a drop-off in performance for payload sizes over 4KB, which is reflected in our results for packets 10KB bytes or larger.

To compare how framework throughput (packets/second) changes when multiple sensors are simultaneously generating data, we ran tests with multiple fake sensors generating 100 byte data packets every 64ms. In this test case, the drivers simply copy the incoming bytes into a parsed sensor reading. The results in Table 6 show that the frameworks perform close to the theoretical max when there are a few sensors running simultaneously. Differences in framework throughput start to emerge as the number of sensors increases. However, we consider 12 sensors to be adequate for all the realistic applications we have considered to date.

Table 6: Framework throughput (pkts/sec) of multiple sensors sending 100 byte packets every 64ms on a Nexus One

Num Sensors	V1	V2	V3	Max
6	93.3	93.2	93.1	93.8
12	186.5	186.4	185.6	187.5
24	371.9	371.8	365.8	375.0
48	737.6	735.2	686.4	750.0

To understand how performance would change on various Android platforms, we measured V2's performance on a few different models of Android devices. Similar to the tests for Table 6, we used 100 byte packets sent every 64ms and varied the number of sensors simultaneously moving data through the driver. As expected, the high-end devices (Samsung Nexus S and Galaxy Tab) had the best performance while the IDEOS had the lowest performance with 48 sensors concurrently running. Overall, the throughput of the V2 framework for 12 sensors is similar across the four devices, confirming that the framework should be portable to a variety of Android devices. While the results presented in Table 7 show all of the devices were able to support multiple sensors with a send rate of 1 packet every 64ms, other tests showed that faster send rates eventually caused errors on all devices. Devices with smaller amounts of RAM and smaller default heap sizes seemed to have more issues. For instance, when delay between packets was reduced to 32ms the IDEOS experienced out of memory errors with 48 sensors, while the Nexus S did not experience errors until 96 sensors were simultaneously sending data. Again, only minor differences were seen when considering realistic numbers of sensors.

Table 7: Throughput (pkts/sec) of V2 framework on different Android devices (100 byte packets sent every 64ms)

Num Sensors	Galaxy Tab	Nexus S	Droid	IDEOS
3	46.4	46.6	44.4	46.3
6	92.8	93.2	92.1	92.2
12	185.0	186.2	184.0	183.8
24	368.6	372.5	366.4	360.6
48	733.7	741.0	720.4	677.4

6.2 Power

The power consumption of each of the three frameworks is negligible with respect to the power use of a device with active Bluetooth or an illuminated screen. We ran 12-hour tests where each framework processed 100 packets/second on a Nexus One. At the beginning of each test, the battery was charged to 100% and each test resulted in an approximate 3% drain of the battery. In these tests, the screen, Bluetooth, Wi-Fi, and cellular radios were off and the only applications running were the battery monitoring application and the sensor framework. These tests show there were only negligible differences between the frameworks with regards to power consumption for expected operating conditions. The tests also show that the power consumption of the framework is negligible in comparison to other phone components.

6.3 Ease of Use/Programming

The ODK Sensors framework provides a reusable code base that makes it easier to create sensing applications by providing a common interface for all sensors (both built-in and external) that abstracts sensor communication details. By decomposing a sensing application into discrete and reusable building blocks, we enable a plug-and-play capability where users with different technical skill levels can contribute in different ways. The system is designed to leverage app marketplaces, which provide a standard interface for users to find and download a variety of applications, making it easy to enhance their Android device to use external sensors. Both application users and developers can leverage marketplaces for distributing and locating sensing applications and sensor drivers that can be reused for a variety of mobile-sensing needs. An application user who wants to collect data from sensors can simply download the ODK Sensors framework, a sensing application (written by an application developer) and the sensor driver(s) (written by driver developers) that corresponds to the sensor(s) they require. An application developer can re-use any existing sensor driver, allowing them to write a sensing application by simply downloading the appropriate driver and interacting with it using the framework's interface. A driver developer's responsibilities are limited to implementing the sensor-specific communication requirements (*e.g.* configuration, start/stop commands, *etc.*). The driver developer can focus on the sensor-specific issues and ignore communication channel specifics of the various wired and wireless channels. A final measure of ease of programming is that the system is portable. In ODK Sensors, all Android devices are interchangeable within the framework as long as the requisite apps and drivers have been installed.

6.3.1 Application User

The application user is expected to be able to obtain and use basic Android applications. To make device driver distribution easy, the drivers in V2 and V3 are designed to be separate applications enabling device manufacturers to post them on their website or an Android marketplace. By using marketplaces such as Google Play, manufacturers take advantage of Android's application distribution system making it easy for users to find and keep their driver updated as the user's device will automatically receive updates (similar to Windows and Mac updates). The framework tries to minimize the work of the end-user by semi-automating the sensor discovery process. The framework automatically discovers all installed sensor drivers on the Android device and populates a list of available devices on each communication channel. The user simply selects the sensing device and is prompted with a filtered list of possible corresponding drivers. The framework's interface also guides the user through any channel-specific pairing tasks (such as on Bluetooth) required to interact with the sensor.

6.3.2 Application Developer

The framework simplifies the application developer's role by providing a single interface to communicate with all external and built-in sensors. This significantly reduces the number of software packages an application developer needs to understand and use in order to communicate with multiple types of sensors over varied communication channels. To further minimize developer responsibilities, ODK Sensors provides a base class for application developers that encapsulates the calls to framework's *Service Interface* to further simplify development by removing the need for the developer to understand Android IPC specifics. Additionally, the framework reduces application developer responsibilities by providing a sensor discovery and driver detection user interface. If an application needs the user to discover a sensor it can send an *Intent* to ODK Sensors to launch the framework's discovery UI. The UI guides the user through sensor discovery, pairing, registering, and driver selection then returns the selected sensor ID to the application, thus eliminating the need for the application developer to re-implement common functionality. If necessary, a specialized user interface could be developed, but at higher development cost.

To quantify the simplification of development we ported four existing standalone sensing applications to leverage the ODK Sensors framework (Section 5). The reduction in the lines of code (summarized in Table 8) in the ported applications provides a basic measure of how development was simplified. We expect standalone applications to have larger codebases because they must implement additional logic to manage communications and handle sensor-specific data parsing.

Table 8: Compares the lines of code needed to create a standalone sensing application vs. an application that uses the ODK Sensors framework. The Sensor Driver column reports the lines of code in the framework's sensor drivers.

Application	Standalone App	App using Framework	Sensor Driver
WaterTime	1350	956	139
Medical	933	246	80
MilkBank	1325	316	105
Built-in Accelerometer	546	538	29

The ported applications leverage the framework for channel and connection management, while sensor-specific data processing is delegated to sensor drivers. Table 9 shows the additional Android modules needed to implement a standalone application that interacts with sensors over Bluetooth, while Table 10 shows the Android modules needed to communicate with a sensor over a *USB Bridge*. Developers leveraging the ODK Sensors interface do not need to understand these Android constructs as the framework hides the internals of these modules from application developers.

Table 9: Additional modules used in a Bluetooth-based sensing application written without using ODK Sensors.

Module	Purpose
Permissions	Allow access to Bluetooth in Manifest
Bluetooth Adapter	Perform fundamental Bluetooth tasks like device discovery and creation of BluetoothDevice and BluetoothServerSocket
Bluetooth Device	Representation of Bluetooth device to create a connection or query information from it
Bluetooth ServerSocket	Listen for connections and create BluetoothSocket to manage the connection
Bluetooth Socket	RFCOMM socket like a TCP socket to allow streaming transport over Bluetooth
InputStream	For input to BluetoothSocket
OutputStream	For output from BluetoothSocket
IOException	Deal with exception on I/O from BluetoothSocket
UUID	Used to initiate an RFCOMM communication
UI Components Handler	To make a UI that will allow a user to discover, pair and connect with a device To receive updates from a Bluetooth I/O thread
Message	Contains data to be sent and handled with the Handler described above when Bluetooth device state changes or data is read / written.
IntentFilter	Used with BroadcastReceiver
Broadcast Receiver	Listen on BluetoothDevice state changes for when a device is discovered

Table 10: Additional modules used in a USB-based sensing application written without using ODK Sensors.

Module	Purpose
Permissions	Set up application to use USB in Manifest
UsbAccessory	Allows you to enumerate and communicate with connected USB accessories
UsbManager	Represents a USB accessory and contains methods to access its identifying information
ParcelFile-Descriptor	Descriptor that can be passed between processes.
FileDescriptor	Descriptor that can be passed between processes.
IOException	Deal with I/O error from USB
FileInputStream	Read from USB
FileOutputStream	Write to USB
PendingIntent	Intents that can be passed to and run by another application.
IntentFilter	Used with BroadcastReceiver
Broadcast Receiver	Used to discover when a USB accessory has been attached

6.3.3 Sensor Driver Developer

The framework shields device driver developers from tasks required for the app to interact with multiple sensors, such as management of channels, threads, buffers, and sensor states. The device drivers in the framework are designed to be stateless processors of data enabling the developer to keep their code simple. A slightly more experienced developer will need to be responsible for the sensor driver implementation because they will need to interpret sensor data sheets to determine how to manage sensor-specific communications, configuration parameters, and data formats. Table 8 lists how many lines of code were needed to create drivers for the application-specific sensors discussed in Section 5. To make driver implementation simple, driver developers can use a base class provided by the framework development team to handle the communication between the driver and the framework. Since the communication version information is contained within the provided base class, the framework will automatically communicate via the correct protocol when discovering the driver, thereby enabling drivers with varying versions of communication base classes to exist simultaneously on the same Android device. This eliminates the requirement of upgrading all driver applications to the newer protocol version simultaneously to updating the framework. Additionally, upgrading the driver to the latest framework communication protocol should be as simple as swapping in the latest binary containing the base class (assuming no changes to the driver interface functions). Driver developers also benefit from the framework's system design of deploying drivers through an Android marketplace. When a device manufacturer updates their driver, they simply need to post the updated application to market and the built-in Android application update system will push out the update to devices that previously installed the driver application.

7. DISCUSSION & FUTURE WORK

To build the next generation of information systems for developing regions, tools are needed to simplify the creation of mobile sensing applications. Barriers to connecting external sensors to mobile devices need to be reduced to enable a wide range of developers with varying programming skills to easily leverage external sensors. While the technical skills required to create and deploy mobile sensing applications are not uncommon in technically astute developer communities, in developing regions the lack of educational resources and technical expertise creates barriers to the development of sensing applications. ODK Sensors' single abstraction for both built-in and external sensors helps developers with basic Android coding skills by reducing the number of Android constructs a developer needs to understand. Additionally, the framework's interface creates a clean separation point between application code and sensor-specific code. This decoupling naturally lends itself to a reusable framework where sensor-specific code is placed into modular drivers. Thus, driver developers can keep their code simple acting as a stateless processor of data. This modular approach also enables different brands/models of sensors to be easily changed by replacing drivers without necessarily requiring application code changes. To further simplify development, the framework provides abstract base classes to make connecting to the framework easier by handling all the inter-process communication. Providing abstract base classes enables easy upgrades to both applications and drivers because the framework developers handle any protocol changes within the base class. The long-term goal of the framework is to enable a market of reusable application components and drivers that can be easily integrated by non-technical users to create sensor-based applications. Deploying mobile sensing applications in under-resourced environments is also challenging because in many circumstances the end-user will likely need to load the sensing software onto consumer devices that may be locked by their service providers. Therefore, for ODK Sensors to be easily deployable by end-users with limited technical abilities the framework and user-level drivers need to fit in Android's app distribution model.

The concept of user-level drivers is well studied and known to have many advantages such as ease of development, portability, and maintainability, but generally suffers in terms of performance. The experiments in Section 6 show the performance of all three frameworks is adequate compared to overall system throughput as limited by the communication channels (Bluetooth and USB). Since framework performance does not significantly impact the system, other factors that lower programming and deployment barriers quickly become more important when choosing the optimal framework architecture. One advantage of the V2 and V3 design is the separation of sensor drivers from the framework thus providing a sandbox environment to minimize any negative effects of misbehaving third-party driver code from the framework. In this respect, V2 and V3 are better framework choices because sensor drivers are isolated as separate Android apps and run in their own virtual machines (VMs). A disadvantage of V2's design is its IPC mechanism acts as a blocking call that can be potentially dangerous if the driver causes the framework to block forever. However, since each sensor is isolated in its own framework thread, the effect of a misbehaving driver on the framework will be minimal as the framework can handle such drivers with a timeout or exception. The main disadvantage to the V3 design is that it is inherently less secure than V2. As any program can register to receive broadcasts making it easy for rogue programs to eavesdrop or inject data. This security problem could be addressed by encrypting data sent between the different processes; however, V3 has other limitations such as timing issues caused by the fact that broadcast messages cannot be received until after the framework's *onCreate* method completes. This method is called when an application binds to the ODK Sensors framework; however, during construction no communication between drivers and framework is possible until after construction is complete because no messages can yet be received by the framework. Unfortunately, once

framework construction completes, applications are capable of sending messages to sensors before the framework's driver connections are properly established causing timing issues.

With sensors becoming increasingly popular on mobile devices, the possibility of multiple applications simultaneously leveraging the same sensor arises and leads to contention for sensor control. Problems can occur when different applications using the same sensor issue conflicting start and stop commands. Additionally, it becomes more difficult for the sensor framework to know how long it should keep a copy of the sensed data cached so that it can deliver the data to all waiting applications. For example, if two applications are simultaneously polling a temperature sensor at different rates, then the framework must maintain data until both applications receive their copy. However, this can be challenging if one application decides it no longer wants the data and sends a stop command while the other application is still waiting. There is a need for techniques to resolve conflicts in resource sharing, leader selection, resource ownership, *etc.* Currently, the framework avoids these issues by only allowing one application at a time to own an external sensor. In the future, we plan to explore models for electing a 'leader' application based on the application that is most dependent on the current stream of data and would be adversely affected by changes to the sensor.

We also plan to continue to simplify sensor integration by expanding the types of sensors that can be connected to the ODK Sensors framework through a *USB Bridge* by including support for a variety of interfacing boards (*e.g.* IOIO, Phidgets). Additionally, as Android devices with USB Master or USB On-The-Go become common, we plan to enhance the *USB Manager* to act both as a USB Slave and USB Master. We are also working on channel managers to manage access to the new class of NFC-enabled low-power sensors that will soon be available.

8. CONCLUSION

The platform shift from traditional PCs to mobile devices with cloud services creates a need and opportunity to integrate these devices (*e.g.,* smartphones, tablets) with external sensors and deploy applications in new settings. To address this, we created an application-level driver framework that enables convenient reuse of sensor-specific code between applications by logically separating the high-level application from the underlying sensor driver. The focus of ODK Sensors is on enabling the integration of data from a variety of sensors over both wired and wireless communication channels. It simplifies application development by creating a single interface that can control virtually any kind of sensor (both external and built-in) and reduces the amount of code needed to access a sensor. Applications that leverage the framework to communicate with external sensors can be implemented in fewer lines of code (Table 8) by removing sensor communication code - on the order of ten fewer Android Java modules (Tables 9 and 10). Additionally, the sensor framework automatically multiplexes the communication channels allowing different types of sensors to be used simultaneously by an application. For example, an application can easily use two USB and three Bluetooth sensors simultaneously to record several phenomena at once. The ODK Sensors framework is designed to flexibly meet any application's needs regardless of data type, data collection rate and size, sensor configuration requirements, or communication channel.

After testing the three framework implementations, it was clear that performance was not the most important factor to consider when selecting the final design, as most sensing applications sample data at a significantly lower rate than the framework's maximum throughput. The performance analysis showed that the system bottleneck is the throughput of the Bluetooth and USB communication channels rather than framework throughput. Since the three frameworks performed similarly, other factors were examined before deciding which is optimal. The V2 framework offered the best tradeoff in terms of programming ease, deployment ease, and performance. The separate driver app design makes it easier for end-users to dynamically add new drivers and V2 has better performance than V3. Performance may become more important in the future to accommodate applications that use high bandwidth sensors such as external cameras for medical devices.

This work is part of the larger Open Data Kit [11] project that seeks to develop a modular set of tools to magnify human resources through appropriately designed technology. One of ODK's strengths is creating information systems that collect a wide variety of data types (*e.g.*, location, images, audio, video, and barcodes) that are difficult to record on paper forms. By lowering the barriers to add external sensing components, we hope to expand mobile data collection applications to include an even richer set of data types. ODK Sensors increases the variation of input data types possible by simplifying access to sensing resources through the creation of a single interface that makes external sensors as easy to integrate as built-in sensors. By creating a framework designed to follow ODK's modular components philosophy, we aim to expand the tool suite to allow end-users to easily augment their Android device with external sensing options. The component philosophy enables easy reuse of sensor drivers that will hopefully lead to an ecosystem of drivers further promoting the creation of novel mobile sensing applications. Using standard Android app distribution channels (e.g. Google Play) will make it easy for users to download functionality enhancements (application-level device drivers) to their unmodified Android OS. This simple method of deployment will hopefully lead to the creation of new sensing-based mobile data collection applications that improve information services in under-resourced contexts that typically lack a rich technology infrastructure (both physically and in terms of expertise).

9. ACKNOWLEDGMENTS

We thank Steven Gribble for his helpful advice on this project. We also thank Yaw Anokwa, Brian DeRenzi, and Mitch Sundt for their insightful feedback on this paper. Finally, we thank our anonymous reviewers, and our shepherd Landon Cox, for their guidance. The material in this paper is based upon work supported by NSF Research Grant No. IIS-1111433 and a NSF Graduate Research Fellowship under Grant No. DGE-0718124.

10. REFERENCES

[1] Android Open Accessory Development Kit. http://developer.android.com/guide/topics/usb/adk.html. Accessed April 2012.

[2] Arduino. http://www.arduino.cc/ Accessed April 2012.

[3] A. Bhardwaj, P. Arjunan, A. Singh, V. Naik, and P. Singh. MELOS: a low-cost and low-energy generic sensing attachment for mobile phones. In *Proc. of the 5th ACM Workshop on Networked Systems for Developing Regions*, 27-32, June 2011.

[4] R. Chaudhri, W. Brunette, M. Goel, R. Sodt, J. VanOrden, M. Falcone, and G. Borriello. Open data kit sensors: mobile data collection with wired and wireless sensors. In *Proc. of*

the 2nd ACM Symposium on Computing for Development, 9:1-10, March 2012.

[5] R. Chaudhri, E. O'Rourke, S. McGuire, G. Borriello, and R. Anderson. FoneAstra: enabling remote monitoring of vaccine cold-chains using commodity mobile phones. In *Proc. of the First ACM Symposium on Computing for Development*, 14:1-9, Dec. 2010.

[6] R. Chaudhri, R. Sodt, K. Lieberg, J. Chilton, G. Borriello, J. Cook, and Y. Masuda. Low-power Sensors and Smartphones for Tracking Water Collection in Rural Ethiopia. *IEEE Pervasive Computing*, (to appear), March 2012.

[7] R. Chaudhri, D. Vlachos, J. Kaza, J. Palludan, N. Bilbao, T. Martin, G. Borriello, B. Kolko, and K. Israel-Ballard. A system for safe flash-heat pasteurization of human breast milk. *In Proc. of the 5th ACM Workshop on Networked Systems for Developing Regions*, 9-14, June 2011.

[8] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma. PRISM: platform for remote sensing using smartphones. *In Proc. of the 8th Int. Conf. on Mobile Systems, Applications, and Services*, 63-76, June 2010.

[9] N. Dell, S. Venkatachalam, D. Stevens, P. Yager, and G. Borriello. Towards a point-of-care diagnostic system: automated analysis of immunoassay test data on a cell phone. In *Procc of the 5th ACM workshop on Networked Systems for Developing Regions*, 3-8, June 2011.

[10] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. *SIGOPS Operating Systems Review,* 42(2):168-178, March 2008.

[11] C. Hartung, Y. Anokwa, W. Brunette, A. Lerer, C. Tseng, and G. Borriello. Open Data Kit: Building Information Services for Developing Regions. *ICTD 2010*, Dec. 2010.

[12] 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. *Wireless Health 2010*, 34-43, Oct. 2010.

[13] Y. Hong, H. K. Worden, and G. Borriello. ODK Tables: data organization and information services on a smartphone. In *Proc. of the 5th ACM Workshop on Networked Systems for Developing Regions*, 33-38, June 2011.

[14] C. Hsieh, H. Falaki, N. Ramanathan, H. Tangmunarunkit, D. Estrin. Performance Optimization of Android IPC for Continuous Sensing Applications. *CENS Technical Report #104*. April 2012.

[15] IOIO for Android. Android Development Tools. http://www.sparkfun.com/products/10748. Accessed April 2012.

[16] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 59-72, Oct. 2009.

[17] A. Kadav and M. M. Swift. Understanding modern device drivers. In *Proc. of the 17h Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 87-98, March 2012

[18] B. Kaufmann and L. Buechley. Amarino: a toolkit for the rapid prototyping of mobile ubiquitous computing. In *Proc. of the 12th Int. Conf. on Human Computer Interaction with Mobile Devices and Services*, 291-298, Sept. 2010.

[19] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5):654-664, Sept. 2005.

[20] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two Years of Experience with a (Mu)-Kernel Based OS. *SIGOPS Operating Systems Review*, 25(2):51-62, April 1991.

[21] F. X. Lin, A. Rahmati, and L. Zhong. Dandelion: a framework for transparently programming phone-centered wireless body sensor applications for health. *Wireless Health 2010*, 74-83, Oct. 2010.

[22] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: using low-power processors in smartphones without knowing them. *In Proc. of the 17th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 13-24, March 2012.

[23] B. Priyantha, D. Lymberopoulos, and J. Liu. Enabling energy efficient continuous sensing on mobile phones with LittleRock. *In Proc. of the 9th ACM/IEEE Int. Conf. on Information Processing in Sensor Networks*, 420-421, April 2010.

[24] M. J. Renzelmann and M. M. Swift. Decaf: moving device drivers to a modern language. In *Proc. of the 2009 Conference on USENIX*, 14-14, June 2009.

[25] D. W. Richardson and S. D. Gribble. Maverick: providing web applications with safe and flexible access to local devices. In *Proc. of the 2nd USENIX conference on Web Application Development*, 12-12, Oct. 2011.

[26] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: hierarchical power management for mobile devices. In *Proc. of the 3rd Int. Conf. on Mobile Systems, Applications, and Services*, 261-274, June 2005.

[27] D. Talbot. Android Marches on East Africa. *Technology Review*, http://www.technologyreview.com/communications/37877, June 2011. Accessed April 2012.

[28] N. Villar, J. Scott, and S. Hodges. Prototyping with microsoft .net gadgeteer. In *Proc. of the 5th Int. Conf on Tangible, Embedded, and Embodied Interaction*, 377-380, Jan. 2011.

[29] Z. Zhuang, K.-H. Kim, and J. P. Singh. Improving energy efficiency of location sensing on smartphones. In *Proc. of the 8th Int. Con. on Mobile Systems, Applications, and Services*, 315-330, June 2010.