Open Data Kit Sensors: Mobile Data Collection with Wired and Wireless Sensors

Rohit Chaudhri, Waylon Brunette, Mayank Goel, Rita Sodt,
Jaylen VanOrden, Michael Falcone, Gaetano Borriello
Department of Computer Science and Engineering
University of Washington, Seattle, WA [USA]
{rohitc, wrb, mayank, rsodt, dutchsct, mfalcone, gaetano}@cse.washington.edu

ABSTRACT
Sensing data is important to a variety of data collection and monitoring applications. This paper presents the ODK Sensors framework designed to simplify the process of integrating sensors into mobile data collection tasks for both programmers and data collectors. Current mobile platforms (*e.g.*, smartphones, tablets) can connect to a variety of external sensors over wired (USB) and wireless (Bluetooth) channels. However, the proper implementation can be burdensome, especially when a single application needs to support a variety of sensors with different communication channels and data formats. Our goal is to provide a high level framework that allows for customization and flexibility of applications that interface with external sensors, and thus support a variety of information services that rely on sensor-data. We use four application examples to highlight the range of usage models and the ease with which the applications can be developed.

Categories and Subject Descriptors
H.4 [Information Systems Applications]: Miscellaneous;

General Terms
Design, Experimentation

Keywords
Mobile computing, smart phones, ICTD, sensing, Bluetooth, Open Data Kit

1. INTRODUCTION
Smartphones are becoming a pervasive computing and communications platform, even in developing countries where they are gradually replacing the past generation of feature phones. These new devices are especially attractive for use in developing regions because they are cost effective (as low as $80 as of this writing) leading to 350,000 Android phones being sold in Kenya during the first six months of 2011 [28]. The ICTD community has done a significant amount of work to leverage the improved capabilities of smartphones in bringing about an information management revolution in developing regions. Efforts by researchers like Hartung *et al.* [15] have utilized the enhanced interaction and general computing capabilities of smartphones to improve infrastructure for information services. One of the main reasons for the success of information systems such as [14, 15, 23] is their ability to lower technical barriers for non-technical users. While these systems have addressed the problem of information collection and distribution, they still largely require users to manually enter data into phones and do not leverage the full range of sensing and communication capabilities of increasingly powerful smartphones.

DEV '12, March 11-12, Atlanta, GA

Recently researchers have begun using on and off-device sensing with smartphones to tackle problems that would have been difficult to address otherwise [3, 6, 12]. Moreover, Google has recently announced the Android Accessory Protocol (AAP) [1] that enables compliant phones to connect to external sensors over USB. In addition, there are APIs to access the Bluetooth and Wi-Fi radios on Android devices. These communication APIs allow Android devices to connect to an even wider variety of external sensors. However, building applications that utilize external sensors is relatively complex from the application developers' perspective. In addition to implementing application logic, developers have to deal with the properties of different physical communication channels and handle sensor-specific data processing. These technical barriers can be problematic in developing regions where technical human resources capable of building such systems are relatively scarce. We hypothesize that these technical barriers prevent sensor-based systems from being deployed in the developing world at a large scale and across a range of application domains.

The ODK Sensors framework reduces the complexity of building sensor-based mobile applications by providing abstractions that encapsulate communication channels and by delineating user-application functionality from sensor communication. By defining key abstractions to isolate common functionality, we aim to separate development concerns to reduce complexity and to simplify the connection of components. The overall goal of this work is to determine the appropriate decomposition of a typical Android sensing application to enable code reuse and lower application development barriers. Decomposing the system into modules enables more effective testing and code reuse improving overall system robustness, which is particularly important for ICTD deployment settings. The delineation of application logic from framework logic leads to a cleaner separation of roles, enabling an application developer to solely focus on higher-level application specific concepts, while a driver developer focuses on creating sensor-specific framework drivers to handle sensor-specific details.

The framework provides a reusable code base that makes it easier to create applications by providing a common interface for all sensors. From a user's perspective, the overall setup for ODK Sensors on an Android device consists of two apps: the *User-Application App* and the *Framework App*. For the purposes of this paper an Android application (something you would download from market) is referred to using the word "app", whereas the

word "application" is used to refer to usage/deployment examples. The *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. For instance, to start a sensor, an app uses a generic framework start command to instruct a specific sensor to start collecting data. This command is sent to the sensor by the framework over the appropriate communication channel. Similarly, when encoded data is received from the sensor, the framework forwards it to the appropriate driver for decoding and then makes the decoded data available to *User-Application App*.

Figure 1 shows the overall setup for ODK Sensors on a phone with the two apps: the *User-Application App* and the *Framework App*. Apart from the setup on the phone, an ODK Sensors configuration consists of multiple sensors, potentially connected over different channels types (*e.g.*, USB, Bluetooth, *etc.*). These sensors may connect to the phone over a USB Bridge that enables the actual physical connection to the phone's USB port (*e.g.*, an Arduino-based interfacing board).

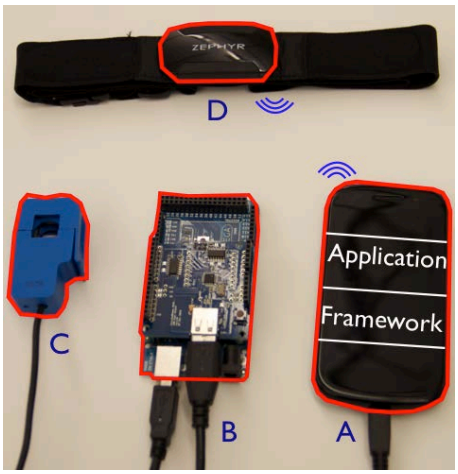

Figure 1: End-user view of the overall setup of an ODK Sensors system. Application and Framework are Android apps that are installed on the mobile device (A). The mobile device is connected to an Arduino interfacing board (B) over USB. A current sensor (C) is connected to the Arduino board's I/O ports. Application asks Framework to connect to and get data from a current sensor and a heart rate sensor (D). Framework connects to the current sensor via the Arduino USB Bridge and the heart rate sensor over Bluetooth, and returns sensor data to the Application.

Consider a scenario where the user wants to track the amount of current flowing through a wire. Using ODK Sensors, the user needs to have the generic *Framework App* and a *Current Monitoring App* that provides the user with the visualization of the data. The *Current Monitoring App* simply asks the framework to connect to a current sensor and fetch instantaneous readings. It does not need to worry about the channel over which the sensor communicates. The framework, on the other hand, handles all low-level communication with the sensor and channel-specific communication protocols. Once the framework receives data back from the sensor, it sends the relevant information in a format and method specified by the *Current Monitoring App*.

This paper presents our work designing the ODK Sensors framework to simplify the process of integrating external sensors with mobile devices like smartphones and tablets, thus lowering barriers for development of mobile sensing applications. To create appropriate abstractions we identify three dimensions of variation in typical sensing use cases and propose a framework that separates development concerns into different roles (Section 2). We then validate the framework decomposition by implementing four representative mobile sensing applications that vary along these three dimensions (Section 3).

2. FRAMEWORK

The framework decomposes a typical mobile sensing application into reusable modules that encapsulate common functionality. This enables development of user apps that focus simply on the overall application logic while the responsibility of processing of sensor-specific data is transferred to driver developers. To assist application developers, the framework provides abstractions to simplify the management of low-level, channel-specific communications, thus creating a clear separation of concerns for developers. The framework was designed for three developer roles:

- an Application Developer who implements top-level user applications,
- a Driver Developer who creates sensor-specific processing and control modules, and
- a Framework Developer who provides the framework itself.

A list of the various modules/components used in the framework is presented in Table 1 along with the type of developer who is responsible for implementing the functionality. The goal of the ODK Sensors project is to shift as much responsibility as possible to the framework developers, simplifying the creation of mobile sensing application. Unfortunately, a framework developer cannot create a framework that can universally communicate with all sensing devices without knowledge of sensor-specific instructions; therefore, the role of driver developer is necessary to create the ability to add new sensing devices to the framework. To encourage new driver development the framework absorbs as much sensor-specific responsibility as possible, including management of connection state and threads. The framework has two main developer abstractions:

- the *Service Interface* and *Content Provider* form the principal unified interface that enables application developers to create apps that use the framework.
- the *Driver Interface* is implemented by *Sensor Drivers* that process sensor-specific data and is only dealt with by driver developers.

The framework supports building sensing applications that vary along three basic dimensions: communication channel, data format, and sensor configuration. The communication channel used by sensors to communicate with the mobile device may vary across sensors and/or applications (*e.g.*, USB, Bluetooth, NFC). Additionally, the type of data collected can vary by format, size, and frequency of data samples. The third dimension is the configuration requirement of the sensor such as sampling rate, trigger conditions or alerts, identifiers, and calibration. Our framework aims to support any combination of communication, data format, and sensor configuration(s).

Component	Description	Developer Type
Android		
Application App	Top-level user app that contains the Application's logic	Application
Service Interface	Entry point into the framework for Application to interface with sensors	Framework
Content Provider	Data store provided by framework for an Application's convenience	Framework
Sensor Manager	Manager that maintains Sensors state and routes messages to/from Sensor Drivers	Framework
Sensor Driver	Module to control a specific sensor and process sensor-specific data	Driver
Channel Manager	Abstraction for physical communication channels (e.g. USB, Bluetooth)	Framework
BT Controller	Object to handle low-level BT communication with sensor	Framework
AAP Communication Layer	Wrapper for AAP to communicate with AAP-compliant devices over USB	Framework
Arduino		
Arduino Controller	Arduino's controller that handles communication over USB and multiplexing sensors	USB Bridge
ARD Sensor Manager	Data structure used for routing messages to/from ARD Sensor Drivers	USB Bridge
ARD Sensor Driver	Module to sample a sensor connected to the Arduino's low-level I/O interface	Driver

Table 1: A summary of Android and Arduino components discussed in this section. The Developer Type column lists the 3 different types of developers responsible for developing components, i.e. Application, Framework or Driver developers. USB Bridge refers to the developer of the Arduino USB Bridge (outside the scope of our framework).

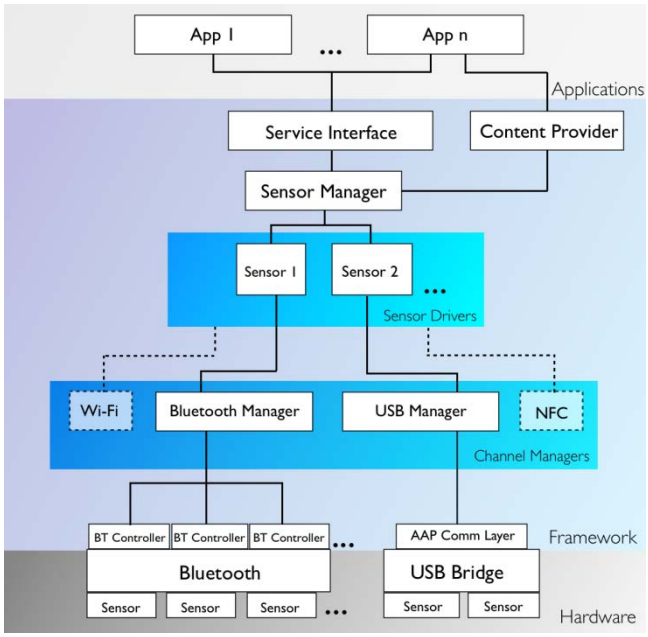

Figure 2: Generic architecture of the ODK Sensors Framework. *Service Interface* and *Content Provider* form the principal unified interface used by apps to communicate with the framework. *Sensor Manager* contains references to all available *Sensors*. *Channel Managers* are specific to communication channels and manage connections and data transfers with underlying hardware sensors.

The ODK Sensors framework (Figure 2) supports multiple underlying communication modalities by providing abstractions called *Channel Managers* that encapsulate channel-specific details such as sockets (for Bluetooth) and handshakes (USB AAP). This encapsulation allows low-level, medium-specific communication protocols to be hidden from both the application and driver developers. *Channel Managers* allow *Sensor Drivers* to be written without knowledge of which communication protocol is being used to transmit the sensor-specific commands. Currently, the ODK Sensors framework has channel managers for Bluetooth and USB. The *Bluetooth Manager* handles low-level communications with all Bluetooth-enabled sensors using Android's built-in libraries. The *USB Manager* handles USB/AAP protocol-specific low-level communications for drivers of sensors that are accessible over USB. Moving forward, the framework will also handle other communication media such as WiFi and near-field communication (NFC).

Additionally, ODK Sensors supports multiple data types, sample sizes, sampling frequency, and a variety of configuration steps by utilizing *Sensor Driver* abstractions that encapsulate sensor-specific data processing. 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*. It converts raw data from the sensor and packages it into `<key,value>` pairs that are forwarded to the app. The `<key,value>` format enables app developers to remain agnostic of the sensor protocol specifics and focus only on sending appropriate commands and handling the collected data. This core functionality (e.g., sensor configuration, data processing) is specific to the sensor type rather than the app. Modularizing the system enables multiple, different apps to interface with the same type of sensor by reusing an existing *Sensor Driver*. In addition to allowing for easy reuse, the *Sensor Driver* design shields the app from changes in the communication protocol, configuration, or data type since these changes are isolated to code that processes sensor-specific data. Shielding apps from these changes leads to potentially more robust systems that are easier to maintain.

Continuing with the *Current Monitoring App* introduced above as an example, assume the analog current sensor requires an ADC (analog-to-digital converter) of a microcontroller to read its value. In our system, the sensor would be connected to an Arduino *USB Bridge* connected via USB to the Android device. When the *Current Monitoring App* starts up, it simply connects to the ODK Sensors framework via the *Service Interface,* sends the sensor's configuration information (*e.g.* sampling rate) to the framework, which routes it to the appropriate *Sensor Driver*. The *Sensor Driver* encodes this information into a format that can be decoded by the *ARD Sensor Driver* and sends it down to the Arduino interfacing board via the *USB Manager*. A driver developer would need to implement the *ARD Sensor Driver* (defined in Table 1) that executes the command and reads the low-level ADC values. These values are forwarded to the *USB Manager* using the *USB*

Bridge. The *USB Manager* routes the data to the appropriate *Sensor Driver* for decoding. The decoded sensor data is made available to the *Current Monitoring App* as generic `<key,value>` pairs via either the *Content Provider* or the *Service Interface*.

2.1 Framework Interface

The modular structure of the framework allows apps to access an assortment of sensors for a variety of use cases via a simple interface. The *Service Interface* is the entry-point for apps to interact with the framework and has the methods shown in the following interface description:

```
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);
}
```

To control a specific sensor, an app references it by its unique sensor ID. If the app already knows the sensor's ID it calls `sensorConnect` to connect to the sensor; otherwise it must go through a discovery process to find the ID. In case the *Sensor Manager* has already discovered and paired with the sensor, it completes the handshake with the device. Otherwise the app launches the framework's built-in, interactive sensor discovery system to find the appropriate driver for this sensor. A mapping of a new sensor ID to the selected *Sensor Driver* is added to the *Sensor Manager* after the discovery process completes successfully. This mapping is used to later route sensor data and app requests to the correct *Sensor Driver*. Discovering sensors and creating sensor ID to *Sensor Driver* mappings at runtime makes the system configurable and simplifies deployments.

The framework provides apps with two mechanisms for receiving sensor data. Apps must specify whether they will use the *Content Provider,* which stores data persistently, to receive data or if they will get data directly via the *Service Interface*, leaving the data storage responsibility to the app. A background thread within the framework retrieves data, at a rate specified by the app, from all *Sensor Drivers* that are registered with apps requiring the *Content Provider* for data delivery. An app that has chosen to use the *Content Provider* queries it directly whenever it needs to get data from a sensor. The data in the *Content Provider* is owned by the app.

2.2 Sensor Driver Interface

The ODK Sensors framework is designed to minimize the programming required to add a new sensor type by centralizing sensor customizations into a single interface that can be utilized by the framework. This is achieved by requiring that all *Sensor Drivers* implement the *Driver* interface shown in the interface description:

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

By forcing all drivers into this abstraction the framework is able to transform high-level application interactions into the lower level of sending commands, receiving data, and parsing data.

To implement a sensor driver, developers need to be familiar with the specifics of their hardware sensor. First, developers need to know what communication interfaces are available for the new type of sensor, which currently is limited to Bluetooth and USB. If the sensor communicates only over a low-level I/O interface (*e.g.* I^2C, SPI, etc.), it will connect to our framework via an AAP-compatible Arduino board. This will require an additional driver on the Arduino to connect to the Android framework over USB. Implementing a driver to run within our Arduino *USB Bridge* is just a matter of inheriting from an abstract base class and overriding pure virtual methods to implement sensor-specific initialization, configuration, and sampling. These drivers are currently developed within the *USB Bridge* codebase, which requires a firmware upgrade when creating or modifying a driver. In the future, we want drivers to be downloaded to the Arduino runtime without requiring a firmware upgrade.

The next important part is to understand the standard messaging protocol of the sensor. The protocol and data format details provided in the sensor's datasheet usually include information about configuring and receiving/decoding data from the sensor.

The framework manages the drivers' state, which makes *Sensor Drivers* stateless. The `configureCmd` method of a driver converts a `Bundle` containing configuration parameters, like sampling rate, into the sensor-specific configuration command, and returns it as a `byte` array. The framework then sends this array to the hardware sensor via the appropriate *Channel Manager*. Digital sensors typically have commands to start or stop data sampling; these commands are affected by the `startCmd` and `stopCmd` methods respectively. The `getSensorData` method implements parsing of low-level, raw sensor data contained in `SensorDataPackets` and returns high-level data contained in `SensorDataParseResponse` that can be easily used by applications. The `remainingData` array passed in to `getSensorData` contains any driver-specific state that is managed by the framework.

2.3 Implementation details

Our framework is implemented on the Android platform, which was chosen because it is open source and supports background processes and rich inter-application communication. The ODK Sensors framework runs as an Android Service that starts (if it is not already running) when an app binds to the service or when an AAP-compliant USB device is plugged into the Android device. The framework has been tested on several Android platforms with varying amounts of memory and CPU power; examples include the HTC Nexus One, Motorola Xoom, Motorola Droid, and Huwaei IDEOs (the low-cost Androids available in Kenya). The framework is implemented in about 3500 lines of code, excluding the sensor-specific drivers. The four *Sensor Driver* implementations are about 100 lines each. To interface with the ODK Sensors framework, apps typically need approximately another 100 lines of code.

The *Bluetooth Manager* handles low-level communications with all Bluetooth-enabled sensors using Android's built-in libraries. It handles multiplexing by creating separate *BT Controller* objects with their own thread to preventing blocking when communicating with each external sensor. The *BT Controller* objects simply handle the RFCOMM socket I/O and dispatch data to the higher-level *Sensor Driver* as a generic `SensorDataPacket` with a payload for sensor-specific data

processing. The *Bluetooth Manager* keeps a one-to-one mapping between sensor IDs and *BT Controller* objects.

The *USB Manager* handles USB/AAP protocol-specific low-level communications for drivers of sensors that are accessible over USB. *USB Manager*, and a wrapper layer over the AAP, handles USB-related events, session management, and messaging. Unlike Bluetooth RFCOMM sockets, multiple sensors can communicate over the same USB channel. So rather than having a dedicated thread per sensor (as provided by *BT Controller*), the USB subsystem has one input and one output thread to handle data traffic to multiplex all USB sensors onto one physical channel.

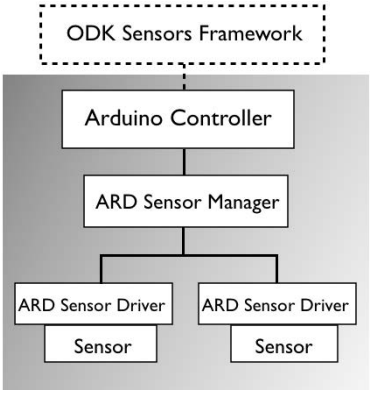

Figure 3: Architecture of the Arduino USB Bridge. The *Arduino Controller* facilitates communication between the Android framework and the *ARD Sensor Manager*, which talks to each sensor through its *ARD Sensor Driver*.

Simple sensors (*e.g.* temperature or current sensors) are typically accessible only over low-level I/O interfaces like I^2C, SPI etc. These sensors are connected to an embedded microcontroller to facilitate data collection. We use AAP-compatible Arduino USB-hosts to connect such sensors to Android devices that implement the AAP (*i.e.* have version 2.3.4 or higher of the Android OS). In our case this excludes the Droid and IDEOs that have version 2.2 of the OS. Our current implementation uses an Arduino Mega2560 as the sensor/Android USB Bridge. Since the Arduino serves as the USB host, it can communicate with all Android devices that have a USB port and run a supported version of the Android OS.

We developed a lightweight framework (depicted in Figure 3) for the Arduino board that implements this bridge. An AAP wrapper class handles USB communications with the connected Android device. Much like the Android framework, the Arduino firmware has an *ARD Sensor Manager* that maintains a mapping of sensor IDs and *ARD Sensor Drivers* that communicate with sensors connected to the board over its low-level I/O interfaces. The Arduino framework is implemented in C++ in about 1000 lines of code. The *ARD Sensor Drivers* implemented so far are about 50 lines of code each.

3. APPLICATIONS

We have developed several applications to evaluate the ODK Sensors framework. These applications were chosen as examples to demonstrate reuse, flexibility, and extensibility of the framework. Three of the four applications shown in Table 2 and discussed below have already been trialed in a developing world context. Leveraging our framework to separate out the application-specific logic from communication logic has simplified the applications significantly from their original form. Building the applications has also helped us understand the interface between sensor drivers and user applications. Together the applications exercise both the wired and wireless subsystems of the framework and, as we see it, are exemplary of the four commonly used styles of data collection in sensor-based systems (Table 2), elaborated in more detail below:

- 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. The medical sensors 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 has an active session with the sensor and observes a stream of samples from the sensor. Monitoring the temperature curve of a milk pasteurization procedure is an example of this use case. The user may want to save a specific window of data for later review as well, as in the case with an electrocardiogram (ECG).
- Snapshot Time-Series: Sensors are deployed to autonomously monitor certain phenomena. They aggregate readings over a period of time and report it to a remote location periodically. Vaccine refrigeration monitoring is an example of this use case.
- Historical Time-Series: Sensors are deployed to autonomously monitor certain phenomena (*e.g.,* movement of an object such as a water can). However, unlike the case of a Snapshot Time-Series, data retrieval is not autonomous and requires human intervention. The WaterTime monitoring application exemplifies this approach.

Application	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

Table 2: Variation in three dimensions (Communication Channel, Configuration, and Data Style) that we have identified as the main differentiating factors of applications as demonstrated by four different sensing applications described in Sections 3.1 – 3.4.

3.1 Medical Sensors

The Medical Sensors application aims to connect a set of tools typically found in a doctor's office with a mobile device making it easy to record medical sensor information along with observational data. In addition to automating modern medical practices, we envision such an application enabling health care workers to take patients' vital readings in ad-hoc locations such as homes, ambulances, rural clinics, and emergency relief stations. Simplifying the operation of medical devices is vital in enabling lightly trained individuals to reliably gather patient data. This will aid in gathering accurate patient records by automating data collection. By recording the data directly on the phone, it can be

used locally as part of a decision support system [13] or sent to a specialist elsewhere for remote evaluation/diagnosis [9]. Further, the application offers the ability to create a mobile diagnostic record of a patient and will eventually be able to interface to electronic medical records systems such as OpenMRS [23].

The ODK Sensors framework allows for any number of sensors to be connected and operated via a single user interface. The Medical Sensors application is designed to support many usage scenarios by providing a framework for interfacing with a common set of instruments that are part of typical doctor's examinations including (but not limited to): stethoscope, otoscope, thermometer, blood oxymetry, and ECG. To enable lightly trained health professionals to become more effective at gathering medical information, each sensor can have an accompanying set of user interfaces to create an integrated contextual help system with educational videos and figures to help ensure proper setup and operation. Projects such as a Midwife's ultrasound demonstrate that assistive UIs for sensing devices can scaffold complex sensing tasks [4]. Typically, during a medical examination there are measurements taken that could be semi-automated with the built-in sensing capabilities of a mobile device. For instance, a mobile phone has a microphone, speaker, a high-resolution camera, and ample data storage. These capabilities make it easier to capture and save measurements and observations. These measurements can be in a variety of formats and may need to be saved for future reference and trend analysis.

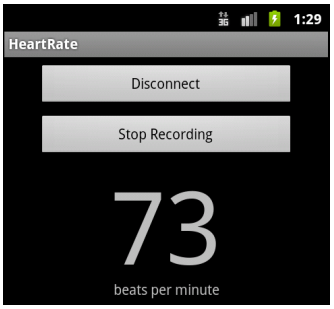

Figure 4: Heart rate application displays the instantaneous beats per minute of a user's heart. This application is an example of single-reading, real-time use-case.

As an example we have implemented a heart rate monitoring app that leverages the ODK Sensors framework to interact with a Bluetooth heart rate sensor. It simply receives instantaneous heart rate (Figure 4) from the sensor and displays it on the UI. This sensor comes pre-calibrated but, if needed, the framework could also be used to calibrate such sensors.

3.2 Monitoring Milk Pasteurization

Breastfeeding is recognized as the best way to get nutrition to a newborn. Breast milk contains the appropriate nutrients to nourish proper development and the antibodies to ensure a strong immunity system. Unfortunately, HIV-infected mothers can transfer the virus to an infant through their breast milk, which is a major cause of child mortality in sub-Saharan Africa [5]. Rather than offering children the much-less than ideal alternative of neonatal formula, milk from HIV-infected mothers can be pasteurized to inactivate the HIV and reduce transmission of the virus. Research shows that HIV can be denatured at a temperature that does minimal harm to the milk's nutrient and antibody content [19]. Flash Heat Pasteurization (FHP) is a low-cost method to accomplish this. However, the process must be carefully monitored.

FHP can be accomplished by heating the expressed breast milk in a jar immersed in a hot water bath. A sensor (stainless steel covered) can be placed in the milk to monitor the temperature curve. FHP heats the milk quickly to 70°C. The person performing the procedure needs feedback that the milk is getting heated at required rate. The milk is then cooled, often frozen, for later use when it is the baby's feeding time. Supervisors want to be able to record the temperature curve to determine that the procedure is being done correctly and to provide additional training.

Figure 5: Flash heat pasteurization setup in the lab.

Chaudhri *et al.* developed a device for this purpose using FoneAstra [7]. Briefly, a temperature probe connected to FoneAstra is used to monitor the temperature of milk as it is heated during the process (Figure 5). Audiovisual feedback from the device through a two-line LCD screen and a beeper guides users during the process and sends the recorded temperature-time curve to a server at the end of the procedure. Feedback from user trials indicated need of a more flexible system that, we hypothesize, would be more practical to implement on a smartphone.

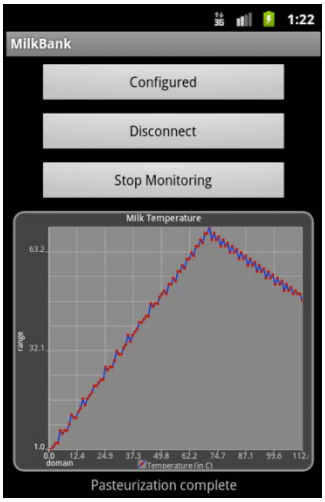

Figure 6: MilkBank application shows the temperature curve during the flash heat pasteurization process. This application is an example of real-time, time series use-case.

Hence we have re-implemented this application to run on Android devices. The top-level MilkBank application (Figure 6) leverages the ODK Sensors framework to communicate with the same temperature probe (used in [7]) connected to Android via an AAP-compatible Arduino board. A 1-Wire temperature sensor driver plugs into the *ARD SensorManager* on the Arduino and is responsible for communicating with the temperature probe. A sensor driver running in the ODK Sensors framework controls the driver on the Arduino. The sensor driver implements the `Driver` interface (described in Section 2) and communicates with the Arduino via the *USB Manager*.

The MilkBank application invokes the framework's `sensorConnect` method to instruct the framework to establish a session with the Arduino driver. Since this is a Real-Time Time-Series application, parsed sensor data is not retrieved via the framework's Content Provider (`getSensorData` is used instead) and the `useContentProvider` flag is set to `false`. The framework's `configure` method is invoked to instruct the framework's sensor driver to configure the sensor, which in turn constructs and returns the appropriate configuration command that is sent to the Arduino driver. The configuration information includes the sampling interval and the number of samples to accumulate in each data-series that is sent back to the Android. For this application the sampling interval is typically set to 1-2 seconds and the data-series length is set to 1. The application invokes the framework's `startSensor` and `stopSensor` methods to instruct the driver to start or stop sampling the temperature probe respectively, which in turn returns the appropriate commands that are sent to the Arduino driver to start or stop sampling. In addition to these simple invocations that set up the driver and sensor appropriately, the application implements its specific logic, which includes periodically invoking the framework's `getSensorData` method to get temperature data from the sensor, providing appropriate feedback to the user based on the current temperature and sending the aggregated temperature data to a server at the end of the procedure.

Separating the application logic from lower-level communications and sensor-specific data processing will make it transparent to support a Bluetooth-enabled temperature probe, if needed at some facilities. Additionally, the rich Android user interface will minimize user-training time.

3.3 Vaccine Refrigerator Monitoring

Vaccine refrigerators throughout the vaccine cold chain must ensure that vaccine doses are kept within strict temperature limits to ensure no loss in potency and effectiveness. Although central repositories might be well staffed and monitored, many district-level and local dispensaries do not have extensive support. Chaudhri *et al.* have used FoneAstra [6] equipped with temperature probes to monitor equipment used to store and transport vaccines. In this application aggregated temperature readings are periodically uploaded to a server via SMS. Additionally, as soon as temperatures deviate from the required range, an SMS message alerts the server, which further pushes out notifications to the appropriate staff for that facility. In this way, a local dispensary manager can be easily notified even during off hours while there is still time to save the vaccine samples.

Field deployments indicated the need of enhanced visual feedback at facilities that store high volumes of vaccines and configurable options for communication (*e.g.* 3G, SMS, Bluetooth). As with the milk pasteurization application (Section 3.2), it is favorable to use a smartphone-based sensing platform that can provide the local user interface for the supervisor, in addition to having a simpler, lower-cost sensing solution like FoneAstra.

The ODK Sensors framework provides a straightforward way to address the new requirements. A basic monitoring device (Figure 7, top & below-left) can be left at the refrigerator to do alarms and remote reporting, while a supervisor with a smartphone can visit the refrigerator, connect to the device, and observe the local temperature data without accessing the server. The detailed information available at this level (Figure 7, below-right) may be much richer than that reported back to the server and provide details such as every time the refrigerator's door is opened. Configurable communication options will help organizations customize deployments according to their needs and budget constraints. For instance, if it's important to avoid the recurring cellular cost, the device can be configured to relay data only over Bluetooth, although this will of course limit the effectiveness of real-time alerts.

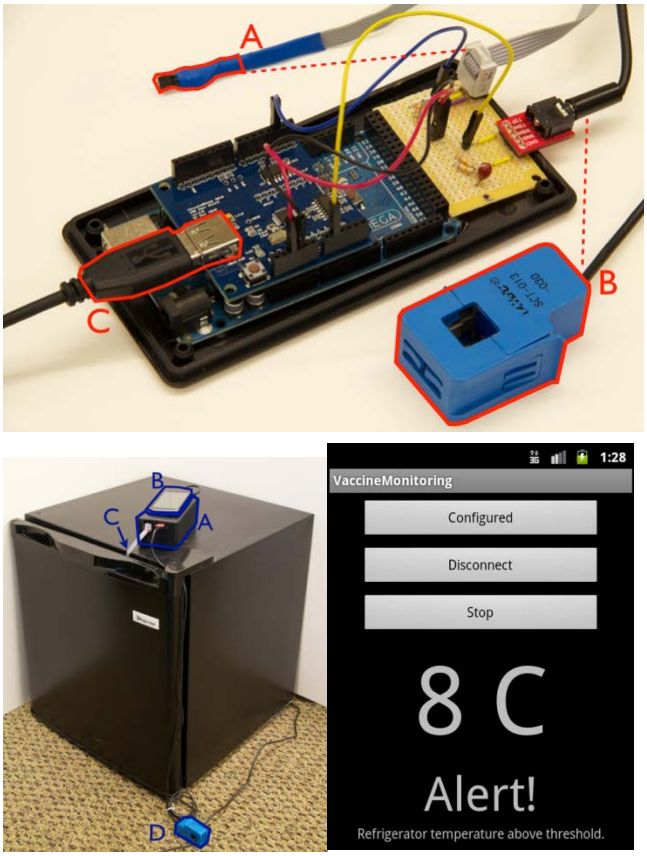

Figure 7: (top) Hardware setup on Arduino board showing temperature sensor (A), current sensor (B), and USB connection to smartphone (C). (Below-left) Vaccine refrigerator monitoring setup showing packaged Arduino box (A), smartphone (B), temperature sensor probe (C), and current sensor (D). (Below-right) VaccineMonitoring is a snapshot time-series use-case.

Since the temperature probe used in the Vaccine Refrigerator Monitoring application is also based on 1-Wire (as in the MilkBank application of Section 3.2), the hardware setup and sensor drivers used on the Android and Arduino are exactly the same as in the previous section. However, driver configuration is slightly different as this is a Snapshot Time-Series application.

The top-level application invokes `sensorConnect` with the `useContentProvider` flag set to `true` as this application retrieves sensor data from the framework's *Content Provider*. The sampling interval typically used is in the order of minutes and each temperature data-series typically has 15-30 samples. The application-specific logic is obviously different, which includes configuring the frequency of reporting to the server, the communication medium used to communicate with the server (3G, SMS, Wi-Fi etc.) and temperature thresholds for alarms.

The MilkBank and VaccineMonitoring applications demonstrate the configurability and flexibility enabled by the ODK Sensors framework to easily support different models of sensing applications, while reusing the same system components.

3.4 Time Studies of Water Gathering

Rural areas in of many parts of the world do not have safe, conveniently located sources of water, so residents spend several hours every day to collect water for household use. Policy makers need to study the amount of time spent by households to collect water (referred to as "time-use") in order to provide convenient water sources (e.g. communal water taps). In a joint-project with public policy researchers at the University of Washington's School of Public Affairs, we developed an application called WuhaGize ("WaterTime") that collects time-use data [8]. WuhaGize was trialed at households in three villages of Ethiopia during the summer of 2011. For this application we developed a Bluetooth-enabled, battery-powered motion sensor (Figure 8) that is attached to jerry cans used for collecting water. Data recorded by the sensors during time-use studies is retrieved by an Android phone over Bluetooth.

The WuhaGize application was developed as a monolithic app that implemented the application-logic and all the low-level interactions between the phone and sensor. To evaluate how well the ODK Sensors framework could handle an existing application, we refactored the original WuhaGize app to create a new app that leveraged our framework. This allowed us to compare a monolithic sensing app with its analogous, decomposed version, built on top of our framework. Additionally, WuhaGize acted as an example of a Historical Time-Series, an additional data collection style that our framework was able to support.

Figure 8: Researcher attaching the WuhaGize sensor, encased in a black box, to a water container. The inset shows the sensor attached to the water container.

Refactoring WuhaGize distributed its functionality into three components that take advantage of the framework's abstractions. The top-level user-app is responsible only for the user interactions and application-specific logic. Sensor-specific logic is implemented in a *Sensor Driver* that lives within the framework. Communications with the sensor hardware are handled transparently via the framework's built-in *Bluetooth Manager*. The user-app communicates with the framework over the unified interface to configure sensors, start data collection, and process the data it receives from the framework, while the framework handles everything else behind the scenes.

4. RELATED WORK

The variety and number of mobile applications has increased in recent years due to the popularity of smartphones and app-stores. However, the number of applications that leverage external sensing devices is limited, in part due to the programming challenges of implementing communication between smartphones and external sensors, and in part due to energy constraints and deployment complexities, which prevent adoption of these applications in resource constrained environments. This leads to two main areas of research: 1) reducing programming barriers [10, 22] and 2) making mobile sensing applications more efficient and simpler to deploy [21 26, 27, 29]. Within these two areas some work focuses primarily on on-device sensors, while others seek to expand communication to sensors not built into the phone. Our work is an effort to not only lower barriers of application creation, but also to provide for a high level of customization and flexibility that increases the variety of external sensors that applications can use.

The Reflex [22] project seeks to bridge the programming gap between simple phone applications and those that interface with external sensors by presenting a framework for transparent programming of heterogeneous smartphones for sensing. They present a suite of runtime and compilation techniques that conceal the heterogeneous distributed nature of the system that results from the realization of methods proposed in Dandelion [21] to reduce power consumption by offloading data processing to low-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. Reflex's concept of a *module* for in-sensor data processing is equivalent to ODK Sensor's concept of a *Sensor Driver*. However, the sensor driver executes within the framework rather than on a separate co-processor. LittleRock [26] and Turducken [27] present architectures that offload continuous sensor data processing to a dedicated low power processor.

Other frameworks have been proposed that are similar to ODK Sensors, but seek to interface primarily with built-in sensors. Zhuang *et al.* [29] introduced an adaptive location-sensing framework that improves energy efficiency of location-based applications through suppression or substitution of location requests from built-in GPS sensors, which represent a significant power drain. This framework is a layer between the application and sensor but instead of lowering programming barriers, it seeks to reduce energy costs. One of the applications that share our framework's goal of lowering barriers for sensing applications is AndWellness [16], which lets researchers customize surveys to collect data from sensors on phones held by participants in their studies. However, unlike ODK Sensors, this application focuses primarily on making life easier for the researcher with customizable surveys and front-end visualization of incoming data

in real-time, rather than on simplifying the programming task for the developer. Our work aims to lower barriers for the application and sensor driver developer and in the future interface with applications such as ODK Collect [15] or ODK Tables [17], which will help make an easy to use program for the end-user.

PRISM [10], like ODK Sensors, is a sensing middleware whose aim is to help developers deploy sensor applications without reinventing the wheel each time or needing to be worried with distributed operation, security, and privacy of those applications. PRISM is evaluated on a variety of applications, but these applications only interface with sensors built into the phone with the focus on deploying them at scale. In contrast, our framework focuses on interaction with external sensors by abstracting away the communication layer to retain programming ease.

IOIO [18] and Phidgets [25] are development boards designed to work with Android phones over USB. They abstract the communication between external sensors and software running on the smartphone, enabling Android applications to directly control hardware attached to them. They are both similar to the Arduino *USB Bridge* in our system. Amarino [20] is another toolkit that connects Android phones with Arduino microcontrollers via Bluetooth. While the goals of Amarino are similar to our Arduino *USB Bridge* (but over Bluetooth), the overall ODK Sensors framework is focused on turning an Android device into a platform that can connect to a wide variety of sensors over multiple communication channels.

5. DISCUSSION AND FUTURE WORK

The ODK Sensors framework is part of the larger Open Data Kit [15] project to develop a modular set of tools to facilitate development of new information services in under-resourced contexts that lack sufficient technological infrastructure and expertise. This project aims to expand ODK by providing a tool to build information services with externally sensed data, thereby removing the need for error-prone manual entry of sensor readings. For the ODK Sensors framework to be utilized in resource constrained environments it will need to connect to an assortment of low cost, off-the-shelf sensors. We plan to expand the types of sensors that can be connected through the ODK Sensors framework by supporting a wider range of *USB Bridge* interfacing boards (e.g. IOIO, Phidgets). Also, as additional Android devices ship with USB-master or USB On-The-Go, we will enhance the *USB Manager* to act both as a USB-slave and USB-master. We also plan to enable access to the new class of NFC-enabled low-power sensors that will be coming to market soon. Finally, we plan to integrate built-in Android sensors into the framework, creating a single interface for all sensing applications. Integrating all the built-in sensors will further verify that the framework interface is sufficiently generic to support a variety of sensor types.

Multiple applications leveraging multiple sensors simultaneously leads to a possible contention for sensor control. This raises the requirement for conflict resolution techniques like leader-selection to resolve configuration differences (e.g., sampling rate). We also plan to explore models of shared ownership of sensor-data, for example determining when is it safe to remove data from the shared *Content Provider*.

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. However, as currently implemented, the framework and *Sensor Drivers* are tightly coupled because *Sensor Drivers* execute within the framework app. Currently, adding a new *Sensor Driver* requires the app to be recompiled, which makes extensibility more difficult than it should be. Our next step is to separate the *Sensor Drivers* from the framework into separate Android apps. We hope this will lead to an ecosystem of driver implementations that can be easily combined under the framework and downloaded from the Android Market just as other apps today.

6. CONCLUSION

In regions where smartphones constitute a significant portion of the computing infrastructure, there is an impetus to adapt mobile devices to leverage sensors traditionally designed for standard PCs or custom hardware. In this paper we presented ODK Sensors, a framework aimed at lowering development barriers for sensor-based mobile applications. This is particularly helpful in promoting ICTD efforts, as highly technical resources are relatively scarce in many parts of the developing world. We identified three dimensions of variability in typical sensing applications, namely: communication channel, data collection style, and sensor configuration.

The ODK Sensors framework creates a clear delineation of developer roles by decomposing a typical mobile sensing application into reusable abstractions and modules. This enables application developers to focus only on the high-level application logic, while driver developers focus on creating reusable modules that handle sensor-specific data processing and control. The framework handles most of the complexities that are common across sensing applications including: communication channel specifics, connection state management, threading, data buffering, etc. The Framework's abstractions isolate changes behind the service interface, shielding top-level apps from such changes. We validated our system decomposition boundaries by building four representative applications that varied along the three dimensions. Decomposing the system into simpler modules also improves the overall robustness of the system because this enables more effective testing and debugging of individual modules. Additionally, as the adoption of reusable modules increases, it also helps build confidence in developer and user communities.

7. ACKNOWLEDGEMENTS

This material is based upon work supported by NSF Research Grant No. IIS-1111433 and a NSF Graduate Research Fellowship under Grant No. DGE-0718124.

8. REFERENCES

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

[2] Basha, E., et al. "Model-based Monitoring for Early Warning Flood Detection". In Proc. of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08), 295-308.

[3] Bhardwaj, A., et al. "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 (NSDR '11), 27-32.

[4] Brunette, W., et al. "Portable Antenatal Ultrasound Platform for Village Midwives". In Proc. of the First ACM Symposium on Computing for Development (ACM DEV '10), Article 23, 10 pages.

[5] Bryce, J., et al. "WHO Estimates of the Causes of Death in Children". Lancet 365. (2005), 1147–1152.

[6] Chaudhri, R., et al. “Pervasive Computing Technologies to Monitor Vaccine Cold Chains in Developing Countries”. IEEE Pervasive Computing. Special issue on Information and Communication Technologies for Development. 2012 (To appear).

[7] Chaudhri, R., et al. “A System for Safe Flash-heat Pasteurization of Human Breast Milk”. In Proc. of the 5th ACM Workshop on Networked Systems for Developing Regions (NSDR '11), 9-14.

[8] Chaudhri, R., et al. “Low-power Sensors and Smartphones for Tracking Water Collection in Rural Ethiopia”. IEEE Pervasive Computing. Special issue on Information and Communication Technologies for Development, 2012 (To appear).

[9] Click Diagnostics: http://www.clickdiagnostics.com/

[10] Das, T., et al. “PRISM: Platform for Remote Sensing Using Smartphones”. In proc. of the 8th Intl. Conference on Mobile Systems, Applications, and Services (MobiSys '10), 63-76.

[11] Datadyne: http://www.datadyne.org/

[12] Dell, N., et al. “Towards a Point-of-care Diagnostic System: Automated Analysis of Immunoassay Test Data on a Cell Phone”. In Proc. of the 5th ACM Workshop on Networked Systems for Developing Regions (NSDR '11), 3-8.

[13] DeRenzi, B., et al. “E-IMCI: Improving Pediatric Health Care in Low-income Countries”. In Proc. of the 26th Annual SIGCHI Conference on Human Factors in Computing Systems (CHI '08), 753-762.

[14] Frontline SMS: http://www.frontlinesms.com/

[15] Hartung, C., et al. “Open Data Kit: Building Information Services for Developing Regions”. In 4th Intl. Conference on Information and Communication Technologies and Development (ICTD 2010).

[16] Hicks, J., et al. “AndWellness: An Open Mobile System for Activity and Experience Sampling”. In Wireless Health 2010 (WH '10), 34-43.

[17] Hong, Y., et al. “ODK Tables Data Organization and Information Services on a Smartphone”. In Proc. of the 5th ACM Workshop on Networked Systems for Developing Regions (NSDR '11), 33.

[18] IOIO for Android: http://www.sparkfun.com/products/10748

[19] Israel-Ballard, K., et al. “Flash-heat Inactivation of HIV-1 in Human Milk: A Potential Method to Reduce Postnatal Transmission in Developing Countries”. JAIDS: Journal of Acquired Immune Deficiency Syndromes, 2007, 318–323.

[20] Kaufmann, B., et al. “Amarino: A Toolkit for the Rapid Prototyping of Mobile Ubiquitous Computing”. In Proc. of the 12th Intl. Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '10), 291-298.

[21] Lin, F., et al. “Dandelion: A Framework for Transparently Programming Phone-centered Wireless Body Sensor Applications for Health”. In Wireless Health 2010 (WH '10), 74-83.

[22] Lin, F., et al. “Transparent Programming of Heterogeneous Smartphones for Sensing”. Rice Technical Report 0310-2011. (2011).

[23] Mamlin, B., et al. “Cooking up an Open Source EMR for Developing Countries: OpenMRS—A Recipe for Successful Collaboration”. In Proc. of AMIA Symposium. (2006) 529–533.

[24] Parikh, T., et al. “Designing an Architecture for Delivering Mobile Information Services to the Rural Developing World”. In Proc. of the 15th Intl. Conference on World Wide Web (WWW '06), 791-800.

[25] Phidgets: http://www.phidgets.com/

[26] Priyantha, B., et al. “Enabling Energy Efficient Continuous Sensing on Mobile Phones with LittleRock”. In Proc. of the 9th ACM/IEEE Intl. Conference on Information Processing in Sensor Networks (IPSN '10), 420-421.

[27] Sorber, J., et al. “Turducken: Hierarchical Power Management for Mobile Devices”. In Proc. of the 8th Intl. Conference on Mobile Systems, Applications, and Services (MobiSys '05), 261-274.

[28] Talbot, D., “Android Marches on East Africa”. Technology Review. MIT. (June 23, 2011). http://www.technologyreview.com/communications/37877/?nlid=4634&a=f

[29] Zhuang, Z., et al. “Improving Energy Efficiency of Location Sensing on Smartphones”. In Proc. of the 8th Intl. Conference on Mobile Systems, Applications, and Services (MobiSys '10), 315-330.