# Open Data Kit 2: Building Mobile Application Frameworks for Disconnected Data Management

Waylon Brunette

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Richard Anderson, Chair

Gaetano Borriello (posthumous), Chair

Kurtis Heirmerl

Beth Kolko

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

**Abstract**

Open Data Kit 2: Building Mobile Application Frameworks for Disconnected Data Management

Waylon Brunette

Co-Chairs of the Supervisory Committee:
Professor Richard Anderson
Computer Science & Engineering

Professor Gaetano Borriello (posthumous)
Computer Science & Engineering

Information technology has transformed the collection, analysis, dissemination, and usage of data. Unfortunately, the variability of available resources, infrastructure, and technical expertise in the world's diverse communities has prevented the digital revolution from benefiting all populations equally. Despite many technological advances, mobile technologies often fail to meet the needs of global development organizations because of poor design assumptions about the availability of infrastructure and resources. For example, the assumption of constant Internet connectivity to access cloud data makes some mobile technologies ill-suited to the long periods of disconnected operation required by many humanitarian organizations. For mobile technologies to be successful in resource-constrained environments they should be usable by minimally-trained users, deployable by resource-constrained organizations, and robust to intermittent power and networking outages. Designing flexible software tools that are configurable by global development organizations necessitates new abstractions that are usable by non-programmers with limited technical expertise. Our research focuses on developing complementary software frameworks, that when used together, create a new generation of mobile tools called "Open Data Kit 2". ODK 2 is a configurable, disconnected data management solution that can be adapted to meet the needs of under-served populations in

resource-constrained environments. Our research aims to design, build, and evaluate multiple software frameworks that create an ensemble of mobile tools that can be used together or independently to produce mobile information systems that can adapt to challenged network environments.

# TABLE OF CONTENTS

# LIST OF FIGURES

# GLOSSARY

ANDROID: An open-source mobile operating system that Google markets and has contributed the majority of the source code. It is primarily designed for touchscreen devices such as smartphones and tablets.

ANDROID APP: Software that is packaged for *end-users* to install Android devices. App are often hosted on Google Play or other software marketplaces. Apps runs in their own process and are made up of at least one of the following four primary Android execution components: Activities, Broadcast Receivers, Content Providers, and Services.

ANDROID ACTIVITY: An Android execution component that is primarily designed to be a user interface to the user enabling interactions with the user. Activities have many built in user interface constructs.

ANDROID BUNDLE: An Android data container construct that is used for inter-process communication. Generally are generally string values enabling parcel-able types.

ANDROID BROADCAST RECEIVER: An Android execution component that receives asynchronous broadcast messages that are sent via Intents from other processes.

ANDROID CONTENT PROVIDER: An Android execution component that stores and retrieves data that is accessible by all applications.

ANDROID INTENT: Asynchronous messages that generally represent an action/operation that an app should performed. Examples usage of intents include broadcast messages, starting services, launching applications.

ANDROID SERVICE: An Android execution component that is used to perform long-running operations in the background that do not require user-interactions. Requests from other other Android processes can be handled concurrently by a threadpool.

APPLICATION: An 'application' in this dissertation refers to an organization using/deploying technology to assist in solving a problem.

CONTEXTUAL DATA CHARACTERISTICS: The characteristics/qualities/properties of a piece of data that are dependent on a application requirements, usage model, or deployment context. The defining trait of an 'contextual' data characteristic is whether the characteristic could change if the subject domain or deployment contexts changes. Examples include data priority, data importance, deadlines, and precedence. Contextual characteristics such as an organization's data policy and local laws can also affect how data is stored and transmitted (e.g., private medical records vs public data).

CSV: Comma Separated Values (CSV) is a method of storing data in a file using a tabular format of data where each set of data is contained on a single line and each individual value in the set of data is separated by a comma.

DATAFLOW: A dataflow specifies how various software tools interact to process and transform data. Dataflows generally create a data pipeline with multiple steps to transform collected data into a format that a user can analyze and consume (e.g., visualization or summary information).

DEPRECATED: A software development term used to describe that a feature, interface, functionality, or piece of software is being retired. The term is used to inform software developers of an upcoming change giving them time to adapt their software to be prepared for the forthcoming retirement and removal. When a software feature, interface, functionality, library, or package is marked 'deprecated,' it is generally regarded as obsolete its usage should be avoided as it will be removed in the future.

FRAMEWORK: A computer programming abstraction that provides common functionality to other software through a programming interface to enable reuse of generic functionalities. In general, frameworks are not modified, but are instead extended or customized through its interfaces.

IDEMPOTENT: This is a property of an operation that implies that the same operation can be applied repeatedly and the value will remain the same no matter how many times the operation is applied. The value will remain the same if the operation was applied once or multiple times.

INHERENT DATA CHARACTERISTICS: The characteristics/qualities/properties of a piece of data that are independent of the application's subject or usage. The defining trait of an 'inherent' data characteristic is whether the characteristic stays the same if the subject domain or deployment contexts changes. Examples of an 'inherent' data characteristic include type and size, as a digital photo characteristic are the same whether

the photo is taken for a medical application or an environmental monitoring application. A digital photo is a binary encoded file of a fixed size of bytes required by the data encoding format.

RACE CONDITIONS: A 'race' condition occurs when the system's behavior is negatively impacted by the dependence on the sequence or timing of events/processes and other events/processes in unknown states. A problem occurs when multiple events/processes need atomic access to a shared resource (such as a database), and multiple events/processes try to gain access to the same resource without having information on when the other events/processes will complete or will need the resource. The term 'race' is used to describe when multiple events/processes 'race' to get access to the same resource and block the system until access the resource is available. This 'race' generally happens when an event/process needs access without knowledge of when the other process/event will release the resource (as if states are known, developers can write code to properly sequence the events). The problem occurs from the waiting event/process block the system as it continually tries to access the shared resource while another event/process has atomic access to the shared resource and does not know it should release the resource.

SQL: Structured Query Language (SQL) is a programming language designed to be used on relational databases. SQL can be used to write complex database queries as well as add, remove, or update data contained in the database.

XLSX: XLSX is the file extension used to specify a file is encoded in the open XML spreadsheet standard.

SMS: SMS is an abbreviation for Short Messaging Service which are more commonly referred to as "text messages". Short Messaging Service enables mobile phone users to send and receive short text messages to/from other mobile phone users.

WEBKIT: A web browser that is embedded into an Android app's interface to enable an Android app to render web pages without the need to use a web browser (e.g., Chrome, Firefox).

WORKFLOW: A workflow specifies how users navigate the software to perform the activity or process required to complete the organization's task. For example, when performing data collection, a workflow defines the order the questions are asked and applies specified logic to adjust which questions are asked based on previously collected data.

# ACKNOWLEDGMENTS

I would like to thank my adviser and Open Data Kit (ODK) project visionary Dr. Gaetano Borriello. Gaetano invited me to co-found the ODK project with Carl Hartung and Yaw Anowka at Google in 2009. I would also like to thank Richard Anderson who became my adviser after Gaetano passed away. Richard played a significant role in helping me to complete this dissertation.

The ODK project has been a team effort that involved many people at the University of Washington (UW). I would like that thank all my co-investigators on the various projects that created ODK tool suites. Specifically, I would like to thank my fellow graduate students who contributed to ODK including Carl Hartung, Yaw Anowka, Rohit Chaudhri, Nicola Dell, Mayank Goel, Matt Johnson, Fahad Pervaiz, Rita Sodt, Sam Sudar, and Morgan Vigil (UCSB). I would also like to thank our software engineers Nathan Breit, Jeff Beorse, Clarice Larsen, Adam Rea, and Mitchell Sundt for their years of extraordinary effort helping to make ODK a reality. Additionally, I would like to express my gratitude to our undergraduate research investigators Marshall Bradley, Nathan Brandes, Malcolm Daigle, Michael Falcone, YoonSung Hong, Ori Levari, Shahar Levari, Li Lin, Luyi Lu, Madhav Murthy, Saloni Parikh, Dylan Price, Clint Tseng, Jaylen VanOrden, and Nicolas Warden.

I would also like to thank Ruth Anderson, Brain DeRenzi, Beth Kolko, Neha Kumar, and Trevor Perrier for giving me advice and helping to shape ICTD research at UW. I wish to thank all of my ICTD labmates at UW for listening to my ideas, encouraging me, and enduring my pontifications. I would also like to thank the many friends I made while in graduate school.

Additionally, I would like to thank all the organizations who used ODK and provided valuable feedback including the International Federation of Red Cross and Red Crescent (IFRC), Jane Goodall Institute, Mercy Corps, PATH, University of California - Berkeley, University of California - San Francisco, UW Global Health, VillageReach, and the World Mosquito Program.

I also wish to express sincere appreciation to the University of Washington, where he has had the opportunity to work with some great people. The material in this dissertation is based upon work supported by Google Research, IFRC, World Health Organization, Nation Science Foundation (NSF) Grant No. IIS-1111433, an NSF Graduate Research Fellowship under Grant No. DGE-0718124, Bill and Melinda Gates Foundation Grant No. OPP-1132099 and USAID Agreement AID-OAA-A-13-00002.

# DEDICATION

This dissertation is dedicated to all my wonderful collaborators and my original PhD adviser Gaetano Borriello, who unfortunately did not live long enough to see me finish my PhD.

# Chapter 1

# INTRODUCTION

Organizations focused on improving education, health, and economic opportunity in under-served communities often operate in areas with limited resources, power, connectivity, and infrastructure. A challenge for many global development and humanitarian organizations is finding 'appropriate' technology designed to operate effectively in these locations where vulnerable populations often require assistance. Mobile devices have become a favored solution for these organizations as they are among the few technologies suitable for use in most parts of the world because of their mobility, lower power requirements, and their ability to connect to the Internet via multiple networking options. Unfortunately, many mobile technologies are designed for use in resource-rich environments only, thereby creating a "digital divide" between people living in resource-rich contexts and those living in resource-poor contexts. System designers need to rethink some assumptions about network connectivity, power availability, user literacy, and device availability. According to a 2016 World Bank report, over 90% of the world's population already lives within mobile coverage areas but less than half the world's population is connected to the Internet [151]. The scenario where billions of people have mobile phones (over two-thirds of the world's population have a mobile phone[151]) but lack Internet connectivity will likely continue for years. Therefore, to address the "digital divide," research needs to be done about how to design technology to operate effectively in remote areas where billions live without Internet connectivity.

Various research projects have focused on extending connectivity by creating alternative networking infrastructures to extend wireless Internet to remote locations (e.g., Google's Loon[56], long-distance Wi-Fi[107], village base stations[66]). However, until global Internet connectivity is available at an affordable price, a parallel approach is needed to create

flexible mobile software technologies that are usable in challenged network environments. This dissertation presents a complementary parallel approach of creating 'appropriate' mobile application frameworks that are designed to operate in any networking environment, including scenarios with long periods of disconnected operation. Even in locations with normally abundant connectivity, networking infrastructure can be damaged or destroyed during natural disasters. To be effective during natural disaster responses and other humanitarian crises, technology needs to be designed to function in disconnected environments for humanitarian organizations to use during a crisis. Unfortunately, many global development organizations still use paper solutions because mobile digital solutions are often designed with poor assumptions about connectivity, available resources, and infrastructure. The unavailability of suitable software tools for mobile data management can be a limiting factor to the scale and complexity of the services humanitarian organizations can provide to beneficiaries. Challenges in deploying applications in developing regions have been documented [17] and include: low literacy, limited technical personnel, and context-specific customization.

A challenge for many global development organizations is finding 'appropriate' technology designed to operate effectively in diverse environments that have varying constraints (e.g., infrastructure, funding, human resources). To enable the deployment of scalable software systems, software developers need to create abstractions that facilitate an organization's ability to customize mobile data applications for use in low-resource contexts. When possible, these abstractions should be designed to empower non-programmers to make customizations to match the organization's requirements. Since requirements can vary based on the local context, organizations need be to be able to customize their mobile data application based on the specific situation characterized by factors such as available Internet connectivity, local languages, data hierarchies, data fields, and organizational workflows. Unfortunately, many existing mobile software frameworks do not prioritize enabling non-programmers to optimize for resource-limited environments. To be flexible enough to meet the needs of humanitarian organizations, the software needs to be designed without a priori knowledge of the exact application requirements or deployment conditions. Therefore, to enable widespread use,

new flexible abstractions are needed to facilitate the adaptability that organizations need to customize their mobile information systems to resource-poor deployment contexts.

The adaptability of a mobile data management application to the requirements of a local context has multiple aspects that often requires an organization be able to customize both the *workflow* and the *dataflow*. A *workflow* specifies how users navigate the software to perform an activity or process required to complete the organization's task. For example, when performing data collection, a *workflow* defines the order the questions are asked and applies specified logic to adjust which questions are asked based on previously collected data. A *dataflow* specifies how various software tools interact to process and transform data. *Dataflows* generally create a data pipeline with multiple steps to transform collected data into a format that a user can analyze and consume (e.g., visualization or summary information). These include abstractions to enable domain-experts to customize to local deployment contexts but also maintain the system's ability to produce an output that is usable for global data analysis, comparison, and action. Additionally, global development funding agencies are recommending that organizations apply evidence-driven development. Evidence-driven development requires organizations to gather data to make decisions, causing information systems to become indispensable for organizations to make informed decisions. Digitizing both the *workflow* and the *dataflow* often improves data accuracy, improves the timeliness of information, increases accountability, and increases data visibility to decision makers. Having reliable data also helps decision makers with monitoring and evaluation, selecting where services should be delivered, planning and managing resources, and comparing various interventions' results.

To empower resource-constrained organizations to build information services that can function in under-resourced contexts, we created multiple suites of malleable mobile data tools called Open Data Kit (ODK) [20, 22, 62]. ODK focuses on leveraging commercially available mobile devices and cloud platforms to simplify an organization's ability to create and scale information systems. ODK's mobile apps are designed to operate on Android-compatible devices because Android devices come in a variety of form factors with different

prices, thus making the devices popular in economically constrained environments. Android is the most common smartphone operating system (OS), with more than an 86% market share in 2019 [119]. For the purposes of this dissertation, an Android application (software package that installs on an Android device) is referred to using the word "app," as the word "application" is reserved for the concept of an organization using/deploying technology to assist in solving a problem.

ODK focuses on providing a suite of interoperable tools that 1) are customizable to a deployment context by a non-programmer and 2) can operate in disconnected environments. ODK's configuration abstractions are flexible enough to handle various field conditions but simple enough for a non-developer to make adjustments as field conditions can and will change quickly. This design requirement differentiates ODK from many alternative mobile application frameworks that are less appropriate because they are designed for use in high-resource environments where skilled software developers can make customizations. Relying on highly-skilled personnel is often problematic in resource-constrained contexts as highly trained personnel are expensive and scarce, leading to situations where organizations struggle to employ technology effectively.

Existing mobile technologies often do not provide adequate interfaces to enable novices to customize the technology to their resource-poor deployment contexts to accommodate limited resources (e.g., battery life, Internet connectivity). For example, when field workers are in remote areas, it can be advantageous to delay certain processing until it can be done in locations with abundant power and low-cost Internet. Additionally, existing mobile frameworks often assume Internet connectivity; while some frameworks offer local data caches, they often lack the full offline replication and synchronization capabilities required to enable field workers to work offline for weeks. Designing mobile data management systems for challenged network environments necessitates creating abstractions for organizations to configure and customize to their deployment context. To address these constraints, we developed modular service-based application frameworks aimed at a variety of use cases, including humanitarian disaster response, public health interventions, and environmental monitoring,

to verify that the frameworks could be utilized without prior knowledge of the application domain.

### *1.1 Multi-perspective design*

To enable ODK frameworks to be configurable by people with different skill levels, we identified four distinct roles of ODK actors that create, configure, extend, deploy, customize and use a mobile data management application. The establishment of these roles came from our multi-perspective analysis [23] used to create framework abstractions capable of adjusting to varying contexts. The four roles are:

1. **End-users** – these are often field workers that have been deployed to a remote location by an organization to perform remote tasks such as providing services to beneficiaries.
2. **Deployment Architects** – these are often employees of the organization who customize the technology and handle organization-wide restrictions and impose constraints derived from deployment considerations. They are the subject-domain experts who ensure that the *workflows* and *dataflows* meet organizational requirements. They are often non-programmers who are responsible for adapting an ensemble of off-the-shelf software to meet an organization's information management needs.
3. **Programmers** – these are skilled programmers who have been employed by organizations to customize the look, feel, or workflow of the ODK frameworks to meet an organization's deployment requirements. Examples include adding new sensors, data input methods, custom prompts, and custom data types.
4. **ODK Framework Developers** – these are the software developers who create the core reusable ODK frameworks that are behind reusable abstractions targeted at the other three roles.

Our multi-perspective design approach is similar to Martins et al.'s [90] approach to coordinating different perspectives for system power. We identified the roles of various

ODK actors that configure, extend, or customize mobile data management applications. As Martins et al. points out *"the user needs to drive"*; claiming: *"1) The OS cannot always know the resource priorities of all applications; 2) applications cannot always know the functionality priorities of the end-user; and 3) users should choose the right level, trading off functionality versus lifetime"* [90]. We differ from Martins et al.'s insights of focusing on a single end-user role by instead splitting the 'end-user' into two roles: *end-users* and *deployment architects*. Additionally, we split the 'application developer' into two roles: *programmers* and *ODK Framework Developers*. The focus on four distinct roles is an important distinction that simplifies the complexity of designing ODK abstractions (e.g., data flows, communication resources, complex workflows). It is challenging to create a single abstraction that is usable by people that have different requirements, different responsibilities, and a wide range of technical skills.

For example, data is often transmitted using whatever protocol a software developer selected when developing the software leaving the user without the ability to modify the transmission based on the context. Figure 1.1 shows how a system can be designed with abstractions to combine information from the network perspectives of the platform (e.g., Android mobile device), software developers (e.g., *ODK Framework Developers*), and *deployment architects* to efficiently manage communication resources to relieve the *end-user* of communication management. Dynamic selection of available protocols based on a deployment's context and a user's location could improve connectivity in challenged network environments by including: 1) metadata from the platform regarding available connectivity; 2) data properties from the software/application developer; and 3) contextual constraints from a *deployment architect*. The *deployment architect* and the software developer (e.g., *ODK Framework Developer*) both provide vital information that is necessary to understand an application's communication context and constraints.

**Platform Perspective -** The platform perspective encompasses the device and operating system perspectives on connectivity and mobility. For instance, an Android device can: detect the type of available network connectivity, detect device mobility, estimate available

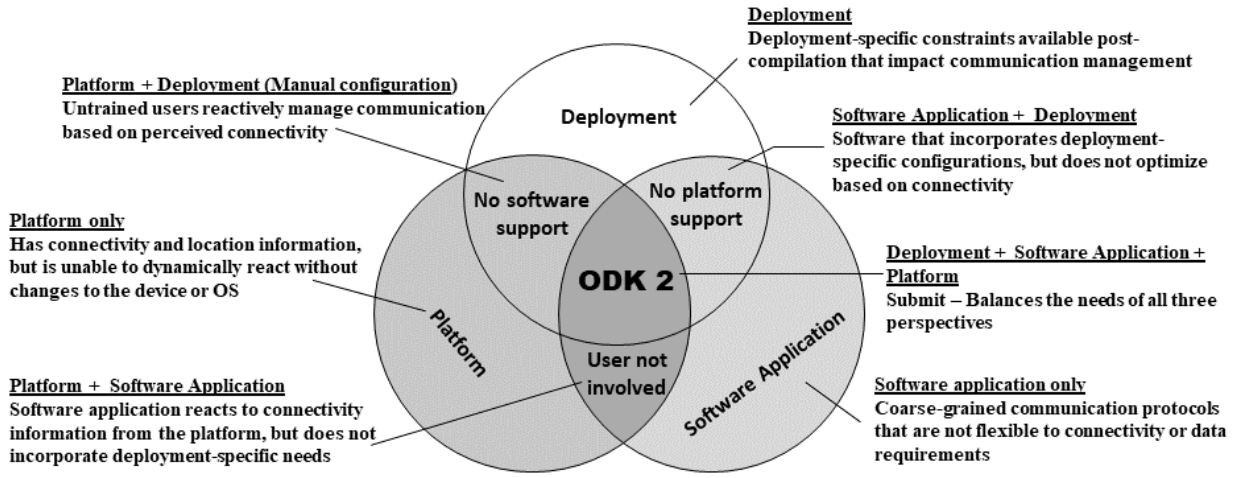



Figure 1.1: Source of information to make design decisions about utilizing heterogeneous networks

data capacity, and estimate the device's geographical location. Location and mobility information can enable the software to infer the duration of a connection and apply regional data policies. However, the platform is unaware of what policies, financial restrictions, and other data priorities a user or organization may want applied to control data transfer decisions. There are numerous related research projects about platform-only communication management schemes, as multiple communication channels can be dynamically allocated based on availability or be bonded to create compound channels with greater throughput capacity [35, 152].

**Application Developer Perspective -** The application developer perspective encompasses issues relating to the functionality of a mobile application. Generally, the software developer understands the *inherent data qualities* such as the type and typical size of data, as data types are determined when writing the software. However, a software developer does not know what the transmission costs to the organization will be to use a particular communication channel. Unfortunately, a developer may be biased towards a particular communication medium and may not bother including functionality to support alternatives

such as: local offline storage to support disconnected operation or transmission of summary data. Thus, developers constrain communication resources via software design and protocol selection. Some examples of developer limitations include apps that communicate over 2G and 2.5G networks exclusively rather than 3G data networks [122, 148]. Likewise, a single protocol limits what an app can effectively communicate, for example, transmitting binary encoded data over SMS is non-optimal. Furthermore, a software developer likely does not fully understand how a future user may want to deploy the application in varying contexts with limiting data policies and budgets.

**Application Deployment Perspective -** The deployment perspective encompasses issues relating to contextual deployment requirements that should be incorporated by a *deployment architect*, as the dynamic contextual information is not available when the developer compiles the software. A deployment's requirements can provide important metadata including information prioritization and financial restrictions. Additionally, this metadata may change during the lifetime of the project, as the deployment evolves. Software frameworks should be designed to be general-purpose and usable in a variety of settings such as public health, environmental conservation, and census applications. These domains have different needs and real-time information may change how data should be transmitted. For example, in a health application, a community health worker may find a patient needing immediate referral to a care facility. This message is urgent and should be sent with a different priority than communicating an update to a healthy patient's medical record. Usage context is not predictable by the developer nor should the platform impose that one particular channel be used, as that channel may not be available. Depending on the urgency of the data it may be necessary to send the same data over multiple channels to ensure delivery or possibly reach multiple destinations.

**Overlapping Perspectives -** There are points where each of these perspectives overlap and interact with another perspective. The most commonly combined perspectives are those of the platform and software application. Opportunistic off-loading approaches combine the connectivity awareness of the platform with software application protocol decisions [10,

59, 84]. While this approach provides more guidance than either the platform or software developer perspectives alone, it results in coarse-grained communication automation. In contrast, web applications exemplify the absence of the platform perspective [98]. While there is value to platform-independent systems, platform information about connectivity and location is necessary for maintaining historical connectivity models.

## *1.2 Global Goods Software*

Building information systems in developing contexts can be challenging because of the limited and diverse infrastructures available in different deployment areas. Many international development organizations integrate *Global Goods* software in their interventions in an attempt to obtain technology efficiency gains observed in other contexts. The aim is to leverage technology to improve process and decision making with an expectation that these improvements will lead to gains in global development outcomes. *Global Goods* software refers to software products that support global development goals. A fundamental challenge of taking *Global Goods* software to an international scale is balancing the different goals and requirements of the different project stakeholders including global organizations that fund projects, the country leadership that controls implementation, and the actual users of the software. To address this, it necessary to have a design process that balances interests of stakeholders and a technical design that allows for modularity and extensibility. The term *Global Goods* [106] is used to identify software products designed for sustainable global development anywhere in the world, with a secondary meaning that there is a public benefit associated with these systems. These systems generally aspire to meet a set of characteristics defined in the *Principles for Digital Development* [110].

Most *Global Goods* projects are motivated by the desire to have a positive impact and advance international development goals. Specifically, we define *Global Goods* as reusable software frameworks that operate with a free and open-source software (FOSS) model. While the FOSS model allows any organization to obtain the software for free, there are often costs associated with deploying and operating FOSS software such as fees for cloud-hosting or

Internet connectivity. *Global Goods* systems are generally supported by communities of developers and implementers with various funding models (e.g., grant-based, consulting fees, hosting fees). Examples of *Global Goods* software include *DHIS2* [14], *OpenMRS* [89], and *OpenLMIS* [144], and *ODK* [20, 62]. Many of these projects aim to achieve global scale to amplify impact and diffuse research and development costs. Additionally, replicating the same software systems in multiple countries increases the efficiency of global organizations' ability to support individual countries.

Many *Global Goods* projects have only reached the pilot stage. The topic of scaling software systems (primarily in the health domain) in low-income countries has been an active topic in the health information systems literature and a source of critiques of the ICTD field [70]. General principles for designing health information systems in low-resource countries have been proposed in multiple publications [79, 88, 95]. The University of Oslo's HISP program has published research on scaling health information systems with papers describing managing standards [13], local problem solving [123], the politics behind system adoption [124], and field deployment studies [14, 15]. One approach proposed by Mwanyika [97] is to define a 'global architecture,' which can then be specialized either as a 'global solution' or a 'country architecture,' to develop a 'country solution.' It is also essential to consider the local obstacles and barriers to taking systems to scale [63, 64].

### *1.2.1 Example Global Goods Systems*

Many FOSS systems could be considered *Global Goods* software; however, this section focuses on software systems that have achieved a significant scale with deployments in multiple countries. To further reduce the number of *Global Goods* systems being analyzed, the focus of the analysis was narrowed to only include systems that have been deployed with the goal of improving patient health (e.g., manage patient records, report treatments, track patient supplies). Specifically, *Global Goods* project categories such as civil registration (e.g., OpenCRVS), health insurance (e.g., OpenIMIS), and human resource management (e.g., iHRIS) were left out of the analysis. Additionally, because of space limitations, representative sys-

tems from different categories of established *Global Goods* software were chosen to represent the category for analysis to focus the analysis on comparing different types of systems.

**DHIS2** – District Health Information Software 2 (DHIS2) is a web-based Health Management Information System [14]. A core functionality provided by DHIS2 is reporting health indicators from facilities and local regions to national regions. Many countries have adopted DHIS2 as their national health reporting tool. DHIS2 is managed by the Health Information Systems Program at the University of Oslo in Norway.

**OpenMRS** – Open Medical Record System (OpenMRS) is a widely used, open-source platform for implementing electronic medical records [89]. The system got its start supporting HIV/AIDS care and treatment and has grown to support primary health care in a broad range of settings. OpenMRS allows custom data structures but uses a common-concept dictionary to link vocabulary and reporting. OpenMRS also supports a software-plugin framework making it easy to add custom modules to adapt to specific use cases. An estimated 3,000 sites now deploy OpenMRS.

**OpenLMIS** – Open Logistics Management Information System is a cloud-based system designed to manage health commodity supply chains in developing countries. The system supports inventory management, ordering, and fulfillment and includes reporting and analytics. VillageReach, a non-governmental organization (NGO), is the leading contributor to OpenLMIS and manages the project as an open-source project [144].

**99DOTS** – 99DOTS was created by Microsoft Research India to perform low-cost monitoring of patients taking tuberculosis medication [32]. Patients make a free phone call each time they take their medication to enable health providers to monitor the patient's adherence to the dosage schedule. 99DOTS involves creating custom secondary envelopes that are packaged around patient medication. This custom packaging hides a series of phone numbers behind the dosage pills. When patients take a dosage, a toll-free number hidden behind the pill is revealed. Patients report taking the medication by contacting the toll-free number that was exposed by removing the medication. 99DOTS relies on most patients having access to a mobile phone capable of making a toll-free call. The reliance on voice

dialing instead of Internet communication has enabled the 99DOTS system to reach a large number of patients since voice dialing is more ubiquitous than Internet application usage. Health workers responsible for monitoring medication adherence have more options, as they can view a patient's medication history via a mobile app, a website, or SMS messages.

**Ushahadi** – Ushahidi, which in Swahili means "testimony," is a crisis-mapping software project developed during the 2008 Kenyan elections to enable people to communicate issues and document post-election violence. Ushahidi continued as an open-source project that makes it easy to crowd-source information so that everyone's voice is heard [140]. Ushahidi is designed to enable individuals or groups to both collect and disseminate information to and from people in the field.

**FrontlineSMS** – FrontlineSMS [50] enables organizations to send SMS messages to collect small pieces of data. Using SMS to collect data can simplify crowd-sourcing data, community communication, and collecting research data. FrontlineSMS makes it easy to adapt to many use cases with features such as automated responses, triggered prompts, and keyword search.

### *1.2.2 Common Features of Global Good Software*

All the chosen representative *Global Goods* software frameworks from section 1.2.1 focus on gathering accurate information and aggregating the data into a usable format that is specific to an organization. The design of these software systems allows organizations to adapt to the local context through the customizability of the *workflow* and/or the *dataflow*. After examining the features of the representative *Global Goods* systems, we found the following common features: 1) customizable to deployment context, 2) modular design, and 3) data update refresh requirements are often long, in terms of hours or days instead of seconds. A comparison of *Global Goods* software features is shown in Table 1.1.

One design decision that a *Global Goods* software project will make is whether to be a subject domain-dependent system or subject domain-independent system. There are advantages to either approach as a domain-independent can be used more broadly by many

Table 1.1: Global Goods Software Frameworks Common Design Feature Comparison [18]

| ***Feature*** | ***DHIS2*** | ***Open-MRS*** | ***Open-LMIS*** | ***99 Dots*** | ***Ushahidi*** | ***Frontline-SMS*** |
|---|---|---|---|---|---|---|
| Scalable | X | X | X | X | X | X |
| Modular Design | X | X | X | | | |
| Localization | X | X | X | X | X | X |
| Abstractions create limitations | X | X | X | X | X | X |
| Subject domain independent | | | | | X | X |
| Subject domain dependent | X | X | X | X | | |
| Flexible data structures | X | X | X | | | |
| Disconnected operation | | X | | | | |
| Health Workers receive data via Internet | X | X | X | X | X | |
| Patients communicate via Voice or SMS | | | | X | X | X |
| Data update/refresh longer than seconds | X | X | X | X | X | X |
| Supports predefined roles beyond data collector and administrator | X | X | X | X | X | |
| Customization abstractions designed for non-programmers | X | X | X | X | X | X |

organizations working in different domains. However, a domain-dependent solution can make it easier for organizations to use by 1) shrinking the number of options that need to be configured for the system to be used, 2) minimizing the vocabulary used to the domain space, making system features more intuitive to users, and 3) having a set of predefined roles (e.g., patients, health care workers, supervisors) beyond that of data collector and system administrator that enable customized interfaces that display appropriate information based on a user's role. However, the scope of the domain-dependence can also vary. For example, both OpenLMIS and DHIS2 are information management systems that target the 'health' domain. However, OpenLMIS is a logistics management information system, whereas DHIS2 is a health management information system. A logistics information management system tracks the number of medicines, vaccines, and other supplies at each facility and assists with ordering and allocating new medical supplies. A health management information system tracks incidences of diseases, health services rendered, and patient information. Another

common design feature addresses the fact that health care workers often need to access larger pieces of data that can be easily be provided by voice (hard to remember 50 numbers) or SMS (messages are limited to 160 characters). Therefore, most of the systems have the "field worker" or "health care" worker send and receive information via the Internet.

### *1.2.3 Fitting within Existing Ecosystems*

A *Global Goods* software system needs to be able to adapt to the existing ecosystems in international development. While technologists generally lead *Global Goods* software projects, there is an implicit requirement that the systems need to be suitable for users in the deployment context. The majority of the representative *Global Goods* software frameworks from section 1.2.1 have adjusted to deployment contexts with evolution in scope and technology. For example, DHIS2 is an outgrowth of work by activists in post-apartheid South Africa [14] who were working to make the health system more equitable. Early versions of DHIS (the predecessor to DHIS2) had failures in several countries, including Cuba [121], before several implementations flourished and achieved a national scale. Both OpenMRS and OpenLMIS have histories with varied types of deployments, with the learnings leading to a significant redesign of their products. The length of time a *Global Goods* project has been active is an influential aspect of fitting into the ecosystem, as it takes time to build expertise and establish themselves in the domain and donor networks.

#### *Software Development Ecosystem*

*Global Goods* systems are generally developed with the goal of achieving social impact and not for commercial reasons. Many of these systems have an academic pedigree as DHIS2 involved the University of Oslo, OpenMRS involved the University of Indiana, and the topic of this dissertation (Open Data Kit) involved the University of Washington. Other *Global Goods* systems were developed by non-governmental organizations (NGOs) to support either internal projects, such as Village Reach developing VRLmis (the predecessor of OpenLMIS), or developed under the direction of a donor, such as iHRIS, which was developed by IntraHealth

under USAID contracts. The supporting ecosystem has passionate software developers, but in many cases, the developers work with less job stability than is provided by the software industry. Many of the contributors to the *Global Goods* software systems often work as either students, open-source developers, or as contractors. From a global perspective, there are often concerns about the code-base of *Global Goods* projects diverging for different countries or contexts, as it creates a challenge to ensure that the many instances are based on a current version of the core software. The evolution of the DHIS project is an example of technology changes affecting *dataflows*. DHIS was initially a Microsoft Access application that relied on 'Feed Forward' files for sharing data. 'Feed Forward' files were appropriate for non-networked PCs of the time; however, when DHIS2 made the shift to a Java-based system, it shifted to using standard network protocols causing the 'Feed Forward' system to become obsolete.

*Funding Ecosystem*

As *Global Goods* software systems are non-commercial, some mechanism is needed to fund the cost of software development and system deployment. Even though the software is free, there are still costs associated with using a FOSS system. There are upfront costs associated with purchasing computing hardware, training users, and configuring the system's *dataflows* and *workflows* so that they are suitable for the deployment context. Additionally, there are costs associated with system maintenance, such as keeping the computers, servers, and mobile devices operating. Furthermore, technology is not static, so as the global technology eco-system evolves, the *Global Goods* system also needs to continue to upgrade to the latest technology releases. To keep FOSS systems operational, someone needs to continually update the system with technology changes and fixes such as operating systems API changes, library bug fixes, and security fixes. Software developers of *Global Goods* projects often identify software maintenance as a particularly difficult area to fund.

Most of the *Global Goods* projects rely on donor and grant funding for core software development activities. This leads to a complex web of financing for systems, including a

mix of funding for core development and deployments. Funding organizations often have a big influence on what features are implemented and what types of behavior will be supported. As donor funding often targets particular health domains, specific use cases are often elevated based on the availability of funding.

*Deployment Ecosystem*

For a *Global Goods* system to be effective, the technologies selected must be suitable for the deployment context. There are standard localization issues (such as language and display formats), along with more complex issues of adapting applications to local *workflows*, *dataflows*, and relevant system outputs. Adapting a system to a local context is often accomplished by providing a customization layer or supporting the inclusion of custom modules. For example, DHIS2 uses an entity/attribute approach to enable the *deployment architect* to specify the health indicators data. OpenMRS utilizes a modular architecture that makes it easy for new modules to be added to provide additional functionality needed for the local *workflow* or *dataflow*. Ushahidi's success comes from the recognition that in 2008 in Kenya the SMS channel of the mobile phone was the best platform to use for crowd-sourcing.

Conditions of receiving funds from donors often lead to certain data standards or other requirements being imposed on countries, thereby often influencing the design of *Global Goods* software. For example, DHIS2 grew based on supporting US PEPFAR reporting requirements as well as countries' desire to report across domains. The need for electronic medical record systems, such as OpenMRS, grew out of funders trying to improve clinical support of HIV treatment. As countries take control of the various health intervention programs, the adoption of systems is dependent on the support of governments that are often concerned about local management of systems and control of data.

## *1.3 Contributions*

Information technology has transformed the collection, analysis, and usefulness of data; however, the variability in resources, infrastructure, and technical expertise across different

communities has prevented the digital revolution from benefiting all populations equally. My research contributes to the design and development of mobile technology frameworks for use in resource-constrained environments. To enable broad use by global humanitarian organizations, I develop software frameworks designed to be adaptable by organizations without a priori knowledge of the application domain, deployment conditions, or how data from various software frameworks will interact.

My research focuses on designing, building, and evaluating multiple mobile software frameworks that provide abstractions that enable users (of varying skill levels) to configure and operate mobile data management systems in resource-constrained environments. Designing mobile applications for low-resource environments necessitates new abstractions that target *deployment architects*, non-developers who adapt software to a deployment context. My dissertation work focuses on identifying software system characteristics needed to achieve global scale and creating frameworks that are adaptable to local contexts. To provide the flexibility that organizations need to adapt software to the local context often requires the customizability of both the *workflow* and the *dataflow*. Data is integral to mobile application design and deployment and has both *inherent qualities* and *contextual qualities* that determine and restrict how data can be transmitted and stored.

For mobile devices to be a useful and effective tool for solving global problems, researchers need to not only expand the capabilities of mobile technology but also expand how mobile technology can be adapted to varying environments with irregular constraints. My research seeks to understand what are the "usable" abstractions that enable users of various skill levels to control "runtime" adaptable systems through configuration files and non-compiled languages. Primarily I seek to understand how to create modular software frameworks that are composable by non-programmers, deployable by resource-constrained organizations, usable by minimally-trained users, and robust to intermittent power and networking outages. Global development organizations need a system that minimizes the user's need to "know how to program" but still enables users to map their workflows, gather data, and digitize their decision making process into a flexible information management system.

My research into information and computing technologies for development has produced the following contributions:

- Identified *Global Goods* software system characteristics needed to achieve global scale and creating adaptable frameworks to local contexts (Section 1.2.1).
- Played a lead role in the creation of ODK's first tool suite, ODK 1, a mobile data collection framework that has been adopted by a wide range of organizations (Chapter 2).
- Gathered data from numerous organizations on the limitations of mobile data collection and management frameworks (including ODK 1). A distinguishing problem was found to exist in use cases where previously-collected data is often revisited and updated (Section 3.1).
- Designed and built multiple modular service-based application frameworks that comprise the ODK 2 tool suite to enhance the ability of organizations with limited technical capacity to build application-specific information services in disconnected environments. ODK 2 is runtime-adaptable to various application domains through the use of automation, configuration files, and runtime programming languages with particular flexibility with regards to data input mechanisms, data models, workflows, and visual appearances (Chapters 3 & 4).
  - Developed *Tables* a data viewing framework that focuses on providing functionality to display, interact, curate, and update the entire data set. Users can explore the data through a variety of built-in views or create custom views using JavaScript/HTML. Reusable templates for viewing and organizing data through the framework's interface make it easier for organizations to design custom views to display and summarize data (Section 4.1).
  - Explored Android design trade-offs in the context of projects in resource-constrained environments to build a *Sensors* framework that simplifies both application and driver development with user-level abstractions that separate responsibilities between the user application, sensor framework, and device driver (Section 4.4).

  - Investigated and built the *Submit* framework to enable organizations to adapt to challenged network conditions such as long periods of disconnection, with a goal of preserving full functionality through local data and device synchronization when connectivity becomes available. This included characterizing various methods of transmitting data and creating framework abstractions to help deployment architects configure their application in areas of sparse heterogeneous connectivity (Section 4.5).
  - Developed a runtime customizable question-rendering and constraint-verification framework called *Survey*. *Survey* provides interactive, non-linear navigation capability and question widgets defined in JavaScript/HTML; thereby, enabling runtime customizable navigation and question data types (Section 4.2).
  - Built a common data services framework called *Services* that abstracts common functionality that is reused by the other ODK 2 frameworks. Examples of common reusable 'services' are: web-server, data synchronization, and a single shared centralized database interface that enables the enforcement of consistency semantics and data-access restrictions (Section 4.3).

- Identified multiple actors in the application space (e.g., *programmer, deployment architect, end user*) that have overlapping concerns but interface with the platform using abstractions of differing complexity. Created abstractions based on this multi-perspective approach that considers various actor's roles and skills that contribute to deploying a customizable mobile data management solution to solve a problem in a specific application (Section 1.1).
- Developed multiple pilot deployments to investigate, test, and verify ODK 2's usefulness and design assumptions (Chapter 5).

# Chapter 2

# THE OPEN DATA KIT PROJECT

Open Data Kit (ODK) is an open-source, modular toolkit that enables organizations to build application-specific information services for use in resource-constrained environments. ODK seeks to simplify the creation of information services by resource-constrained organizations with limited technical capacity. ODK is an extensible suite of open-source tools that supports the convergence of mobility and computing to create an information management system that addresses a variety of digital data needs for global development and humanitarian organizations. Resource-constrained environments often do not have enough technical personnel to build and customize information systems. ODK is designed to be domain-independent and has been used worldwide in a diverse set of domains, including public health interventions, disaster response, election monitoring, documenting human rights abuses, and carbon credit markets. This chapter discusses the early stages of the ODK project development.

Microsoft Office is an exemplar of a suite of software tools that is flexible to many different "applications" and is domain-independent. Global development organizations often use productivity software (e.g., Excel, Word) to create solutions because they can be customized by staff having little programming expertise. While mobile devices are well suited to scarce connectivity and sporadic grid power situations, software applications for mobile devices do not yet offer the same range of features as conventional personal computer productivity software. Mobile apps trend towards having a single purpose and generally are not designed to accommodate a broad range of use cases as compared to a domain-independent application, such as Microsoft Office. Instead, users often use multiple small apps focused on specific tasks with minimal customizability, making it challenging to create custom templates such

as those that are common in Microsoft Word or Microsoft Excel. Users often must leverage multiple special-purpose apps to piece together enough functionality to meet their needs. Mobile frameworks, such as Open Data Kit, are needed to help organizations have configurable mobile frameworks that can be used together to create mobile data management systems. These mobile frameworks need to be able to interact on the mobile device without the need for an Internet connection to mediate the data exchange between different mobile apps (similar to the combined power of Microsoft Office features on a desktop).

A modular design enables organizations to have the flexibility and adaptability to compose a customized information system by using different modular components that best fit the local context. To encourage the creation of multiple system modules, the ODK project uses open standards with a permissive, open-source software license to enable anyone to improve the system. By being open and modular, the barriers to contribution are lower, allowing developers to build new functionality on top of ODK by reusing most of the ODK framework's functionality, saving time and resources. Additionally, using a free and open-source software model allows organizations with limited resources to use ODK software at little or no cost.

The ODK project leverages two computing trends: capable mobile client devices with rich user interfaces and cloud-based scalable data collection, computing, and visualization services. Through ODK, we aimed to create an evolvable, modular toolkit for organizations with limited financial and technical resources to use to create data collection and dissemination services. ODK targets Android-compatible devices because Android is the most popular OS with 86% market share in 2019 [119] and comes in a variety of form factors with different prices. Additionally, Android's flexible inter-process communication methods in 2009 enabled ODK to leverage functionality from existing Android apps to take pictures, scan barcodes, and capture locations.

## *2.1 Open Data Kit Design Principles*

A key design philosophy of the Open Data Kit project is a focus on solving technology issues to help people leverage technology to perform tasks more effectively. ODK seeks to create mobile information systems that "magnify human resources" through 'appropriately' designed technology. By lowering technical barriers, ODK aims to expand people's ability to leverage technology to improve efficiency, accuracy, and robustness, in effect "magnifying" a person's ability to accomplish a task. Therefore, ODK's overall goal is to "magnifying human resources." The idea of technology as a tool that magnifies human intention is analogous to Toyama's 'technology as an amplifier' theory [136]. To be able to "magnifying human resources," my dissertation research combines consumer devices, sensors, and reusable software services to create computer systems that integrate seamlessly into a user's environment. The goal of "magnifying human resources" with technology is similar to Mark Weiser's observation of technologies: *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it" [149].*

ODK's development was guided by a few simple principles [22], notably:

- ***Modularity:*** *create composable components that could be easily mixed and matched, and used separately, or together;*
- ***Interoperability:*** *encourage the use of standard file formats to support easy customization and connection to other tools;*
- ***Community:*** *foster the building of an open source community that would continue to contribute experiences and code to expand and refine the software;*
- ***Realism:*** *deal with the realities of infrastructure and connectivity in low-resource contexts and always support asynchronous operation and multiple modes of data transfer;*
- ***Rich user interfaces:*** *focus on minimizing user training and supporting rich data types like GPS coordinates and photos;*
- ***Follow technology trends:*** *use consumer devices to take advantage of multiple suppliers, falling device costs, and a growing pool of software developers.*

It is also necessary to have a design process that balances interests of stakeholders and a technical design that allows for modularity and extensibility. Part of ODK's success comes from designing reusable frameworks in the local context with *ODK framework developers* working side-by-side with the *deployment architects* and *end-users*. This interaction helps provide *ODK framework developers* real-time, culturally appropriate, in-situ feedback that would not generally emerge without a design partnership with actual users.

## *2.2 ODK Tool Suite 1*

The name Open Data Kit (ODK) refers to the multiple tool suites that are comprised of modular tools. Each tool in the suite has been assigned a name that describes its function. Its modular components are designed to leverage open standards to enable composable software frameworks. The ODK project is designed as multiple configurable frameworks that have public application programming interfaces (APIs). Public APIs and modular design were key design goals used to avoid creating proprietary monolithic solutions by enabling interchangeable modules that could be swapped to provide different functionality. The first tool suite called "ODK 1" [62] was released in 2009 and has been deployed by a wide variety of organizations. ODK 1 provided the ability to design forms, collect data on mobile devices (e.g. mobile phones, tablets), and organize data into a persistent datastore where it can be analyzed. Millions of users have performed data collection activities in almost all of the world's countries. ODK 1 was designed to replace and enhance paper-based data collection, so it focused on collecting data in a unidirectional data flow (similar to paper) without providing functionality to synchronize data back to mobile clients for users to review and update. In the beginning, the ODK 1 [62] tool suite consisted of three primary tools:

- ***Build*** – *Build* is an HTML5 [48] web form designer with a drag-and-drop user interface that generates the user-specified survey logic as an XForm. *Build* is designed to be used by *deployment architects* to create XForm specifications.

- ***Collect*** – *Collect* renders forms into a sequence of input prompts that apply form

logic, entry constraints, and repeating sub-structures. *Collect* is an Android app designed for *end-users* to navigate a wide variety of input prompts (e.g., text, number, location, multimedia, barcodes) to record data. *Collect* is designed to operate by default disconnected from the Internet. *End-users* work through the input prompts and can save the data collected at any point. When connectivity becomes available, users can submit their work to an API compliant server.

- ***Aggregate*** – *Aggregate* provides a server solution that supports data upload, storage, aggregation, filtering, and transfer. *Aggregate* can be deployed on commercial cloud-computing platforms or on local dedicated servers using open-source software. It provides an easy-to-deploy server repository to: manage collected data, provide standard interfaces to extract data (e.g., spreadsheets, queries), and integrate with existing web services as a store and forward system. The goal of *Aggregate* was to store collected data and allow the data to be copied to other data analysis tools. *Aggregate* was designed to be used by *deployment architects* to establish their custom *dataflows*.

These primary tools provide the ability to design forms, collect data on mobile devices, and organize data into a persistent datastore where it can be analyzed. To collect data, *Collect* renders a survey for end-users to gather data. Surveys in ODK 1 are specified using a variant of a W3C XForms standard defined by the OpenRosa Consortium [150]. The XForm contains the data definitions and logic (input constraints and navigation rules) used by the ODK 1 tools. Although XForms can specify input constraints (to provide some immediate error checking abilities), form navigation logic (branching based on previous answers), and multiple languages (for local customization), it does not describe the visual presentation of the prompts and data types. This inability to customize the visual presentation led to many specializations of ODK for different organizations. In ODK 1, the *dataflow* is unidirectional; blank forms flow from the cloud service (*Aggregate*) to mobile devices, and data from the filled-in forms flows to the cloud service and then out to remote services or into file exports. Collected data can be deleted, but is otherwise immutable and provides a store of record.

Data is stored (aggregated) in the cloud, where simple curation and data visualization tools are provided.

User experiences from ODK 1 deployments [62, 22] demonstrate that although non-technical users are able to make small customizations to existing XForms, creating an entire XForm from scratch is often too challenging for them. Thus, *Build* was created to shield users from the complexity of writing XForms by allowing the users to graphically compose surveys. However, some users found the graphic interface too cumbersome for creating complex XForms and for sharing an XForm design so that another user could easily reuse portions. To solve this problem, 'pyxforms' was initially created by Columbia University to give users the option of writing their survey in an Excel spreadsheet that is automatically converted to an XForm. Responsibility for 'pyxforms' was given to the ODK development team to maintain and expand. It became part of the ODK tool suite and was renamed to *XLSForm*.

### *2.2.1 Aggregate*

To simplify the distribution of forms to mobile devices, the retrieval of data from devices, and storing and managing data, we designed *Aggregate*, an auto-configuring, ready-to-deploy server. By providing many forms of data export, *Aggregate* bridges the gap between mobile data collection tools and the sophisticated data analysis software that is capable of deriving complex results. My largest contribution to the ODK 1 tool suite was the creation of *Aggregate*, an automatically configuring server that addressed barriers faced by less-technically capable users when trying to leveraging the power of the cloud. To respect the constraints of organizations with limited financial and technical capital, *Aggregate* was designed to lower the barriers to effective data storage. There are three important barriers that *Aggregate* addressed:

1. Automatically creating the data tables required to aggregate the data that will be collected using a user-designed survey;

2. Providing easy to use filtering and visualization on collected data that resides in the repository; and
3. Connecting to existing external datastores so that collected and filtered data can be incorporated into existing visualization and report-generation tools

To automatically generate the data tables, *Aggregate* uses the same XForm used for data collection to generate the needed database structures. Data types and complex structures (such as repeating sequences of questions) are handled directly, thereby eliminating the need for the user to be concerned with how data will be stored and linked across multiple tables. *Aggregate* enables the user to quickly develop composable filters that can be used to hide unneeded data and/or select specific subsets based on selection criteria. These filters can be saved and reused. The tool also provides the ability to browse thumbnails of media data (e.g., photos) and can generate map overlays for data collection that include geo-coordinates. *Aggregate* can act as a switchboard to channel filtered data to other services so that users' existing tools can have access to the data stream. This facilitates the usage of existing or richer report-generation and visualization systems that provide additional features not found in *Aggregate*. *Aggregate* manages collected data, provides interfaces to export the aggregated data into standard formats (e.g., CSV, KML, JSON) and allows users to publish data to online services (e.g., Google Sheets or Fusion Tables). Thus, *Aggregate* is meant to help bridge the gap between mobile data collection tools and the sophisticated data analysis software needed to derive complex results.

*Aggregate* runs on a Java-based web server and supports multiple databases, meaning it can be deployed to cloud service providers or to a private server. *Aggregate's* design as a 'container-agnostic' platform enables it to be deployed by a variety of organizations that have different infrastructures available. *Aggregate* is a configurable, generic data storage service that runs on a user's choice of computing platform (cloud-based or private server). *Aggregate* can be deployed to the Google AppEngine hosting service to enable a highly-available and scalable service that can be maintained by users who may be less-capable when it comes

to managing information technology. Additionally, some ODK users have data locality and security concerns, either because the data cannot legally leave the country of origin, or because the data may contain sensitive identifiable information, or be high-risk or high-value data. For these users, AppEngine may not be appropriate. *Aggregate* can therefore also run within a Java web container (e.g., Tomcat) using a MySQL or PostgreSQL datastore. Communications security generally relies on HTTPS connections between client devices and the server. However, because many organizations do not have the funds to purchase or the expertise to install SSL certificates on their own servers, we provide user authentication and data security over HTTP communications through DigestAuth and the asymmetric public key encryption of form data before transmission to the cloud. If asymmetric public key encryption is used, the form data is stored in encrypted form on the server, which enables some organizations to continue to leverage cloud hosting services while implementing stronger data security requirements. In this case, users download the encrypted data to a computer and use a locally-running tool called ODK *Briefcase* to decrypt it using a private key.

To provide datastore independence, *Aggregate* incorporates a dynamic datastore abstraction layer rather than a layer set at compile-time. A dynamic datastore abstraction layer was also required to map the document-design style of XForms that is unknown at compile time into column values (to better support filtering and visualization). As discussed in Section 1.1, the *ODK framework developer* does not know what the datastore should look like because the *deployment architect* will specify the data to be collected sometime in the future. Since XForms can define arbitrarily deep nested groupings of repeated questions, *Aggregate* performs a complex mapping of the XForm to a set of database columns and tables. The mapping of arbitrarily-deep nested documents to database tables greatly complicates the processing and presentation of the data. Enabling arbitrary XForm design creates a wide variety of possible data output designs that should be based on the envisioned use case. Unfortunately, arbitrarily deep XForm documents prevents the creation of an easily understandable method for a non-programmer to configure generic processing of the nested repeating sections when visualizing, publishing, or exporting the data. Users were more eas-

ily able to handle tabular forms of data which is why *Aggregate* parses the submitted form instance into column values.

## *2.3 Related Work*

There is an assortment of information and communication technology for development (ICTD) research [16] that demonstrates the potential of technologies to 'magnify human resources' in global development activities [52, 137, 32]. We focus this related work discussion around three primary themes: data collection, field worker perspectives, and abstractions for reusable platforms.

### *2.3.1 Data Collection*

Many organizations (e.g., rural health providers, government agencies) have historically used paper and lack the infrastructure to effectively collect digital data. Gathering accurate information and aggregating data quickly is essential for organizations to have timely and sustainable impacts in addressing their problem areas. Unfortunately, paper systems can be ineffective (error prone, long delay from collection to use, hard to query or aggregate) so ICTD research has produced multiple solutions to digitize data collection and management. Some solutions focus on transcribing paper to digital data. One such project is Shreddr [28], which takes digital images of the paper and extracts data from the paper forms via crowdsourcing digitization. Shreddr was commercialized as Captricity (which is now part of the Vidado AI platform[142]). LocalGround [141] is another example of paper-to-digital tools used to digitize data from manually annotated paper maps to digital maps. Neither of these paper-to-digital tools is designed to work in a completely disconnected operation, which can be problematic for some global development organizations who work offline for weeks. Another approach has been to use SMS for data collection, some examples include FrontlineSMS [50], RapidSMS [115], and UjU [148].

Similar to ODK, a variety of solutions replaced paper-based data collection with digital tools. Two early examples of data collection systems designed for mobile phones were MyEx-

perience [49] and CAM [103]. MyExperience was designed to collect survey responses when triggered by sensor events to document 'in situ' events and feedback from users and was not designed for the configurable organizational information workflows and decision making that many global development organizations desire. CAM [104] is one of the earliest mobile application frameworks specifically targeted at resource-constrained environments. CAM used J2ME phones with a custom scripting language and barcodes to augment paper forms and trigger custom prompts for manual data entry. Since CAM was tied to a specific phone model, the custom scripting language was seen as a barrier to entry as it was a new skill someone had to acquire.

Another early example of a mobile data collection in a resource-constrained environment was the e-IMCI project [42]. e-IMCI used PDAs as the mobile computing device to encode the World Health Organization's Integrated Management of Childhood Illness (IMCI) [101] protocol. While e-IMCI was not customizable by a non-programmer, it demonstrated how mobile computing platforms could improve an organization's workflow in remote locations. The e-IMCI project was an influential project that many researchers (including myself [54, 55]) attempted to further verify and improve by conducting new experiments and deployments [96, 93, 94, 129].

### *2.3.2 Field Worker Perspectives*

Mobile frameworks have the potential to 'magnify human resources' and improve field worker efficiency when they are sufficiently tailored to a deployment context, but they are rarely the only element of a solution. Mobile technology is no panacea and some technology interventions fail because of cultural, organizational, or other issues. For example, Pal et al. discuss the perils of deploying technology without adequately gaining the buy-in from the field workers based on their study of community health workers transitioning from paper to digital data collection [102]. Based on timed observations, the study found that the efficiency and quality of data collected with the paper and mobile tablets were roughly comparable, meaning from the field workers' perspective there was little value to switching. However,

similar to Ramachandran et al. [114], the use of the tablet increased health worker's feelings of empowerment. Pal et al. conducted a second round of research approximately four months later and found that all the health workers had reverted to using paper because of their organizational structure and training. Additionally, health workers struggled to deal with the language barriers of non-localized technology. One limitation of this study was that each participant was only timed completing five paper forms, five tablet forms, and five mobile phone forms, which means that the workers were being timed on their normal method of data entry versus a new method of data entry, which often improves with repetition. In contrast to Pal et al., Perri-Moore et al. [108] conducted a randomized cluster experiment in Dar es Salaam, Tanzania, that showed that health workers using a digitized version of the IMCI protocol on a mobile device significantly improved their counseling performance compared to providers using a paper-based version of the IMCI protocol. Health workers using the mobile devices 1) counseled the mother significantly more frequently, 2) gave significantly more advice on when to return if conditions worsen, and 3) gave significantly more medication instructions [108]. DeRenzi et al. [40] identify advantages and challenges of mobile device based data collection and outline six health system functions that could be supported with mobile technologies: 1) data collection, 2) training and access to reference material, 3) facilitating communication, 4) providing job aids and decision support, 5) supervision, and 6) promoting healthy behaviors. These are organizational goals that try to improve field worker performance.

### *2.3.3 Abstractions for Reusable Platforms*

To assist organizations that often struggle to design effective workflows that incorporate technology, Sudar et al. introduced a conceptual framework called DUCES [133] for characterizing mobile deployments. DUCES classifies the design of a mobile deployment using five axes. These axes are useful in describing deployments and evaluating how well a mobile frameworks abstraction will enable a variety of use cases. DUCES five axes of design are: 1) *dataflow* (unidirectional vs bidirectional), 2) user interface (form-based vs non-form-based),

3) connectivity (connected vs disconnected), 4) edit mode (non-transactional vs transactional), and 5) server model (bucket-based vs processed) [133].

COCO (connect-online, connect-offline) [128] is a web-based data input framework for organizations working in challenged network environments built by Digital Green [44]. Similar to ODK, COCO is a framework that is designed to provide data tracking and analytics to organizations that may have field workers deployed to areas where Internet connectivity is poor or intermittent. COCO is built as a standalone HTML 5 [48] web-application that only requires an Internet browser to operate. Another example of a reusable framework/platform is Apache Cordova [4] (the open-source version of Adobe's PhoneGap[1]). Cordova is an open-source framework for building mobile applications using HTML and JavaScript. To access hardware features of the mobile device, Cordova provides a native plugin-framework that many mobile application frameworks wrap and augment in other products, such as Ionic [72] and Intel XDK [71] (which is now deprecated). Cordova has a different overall focus than ODK as it seeks to create a framework to let programmers write code once and deploy the code to multiple operating system platforms (e.g., iOS, Android, Windows, BlackBerry, Ubuntu, FireOS). Cordova is designed for software developers, whereas ODK focuses on making abstractions simpler for non-programmers to create mobile applications specifically for the Android platform.

# Chapter 3

# ODK 2

This chapter discusses how feedback gathered from ODK users made it clear that there were some deficiencies in our initial design that needed to be addressed. While ODK was successfully used by some organizations to build application-specific information services in resource-constrained environments, other organizations had problems adapting ODK to meet the required *workflows* and *dataflows*. Initially, we sought to incorporate new tools to handle the deficiencies in the original ODK tool suite. This chapter describes the tool deficiencies, the rationale that drove a redesign of the inter-tool architecture of the initial tool suite (ODK 1), and the creation of a new ODK tool suite that was released under the ODK 2 label. Specifically, this chapter introduces ODK 2, discusses the requirement gathering process, discusses related work, and presents case studies that helped shape the design of ODK 2. The overall design of the ODK project (including both tool suites) still focused on creating software tools that ran on Android mobile devices and various cloud services in order to be widely usable in spite of the information technology gaps that exist in under-served regions.

## *3.1 Requirements Gathering*

The ODK 1 tool suite is a successful data collection platform used by many organizations to digitize their data collection in the field. The system design focused on collecting data via digital surveys that are then aggregated in the cloud or on a PC for analysis. However, ODK 1's purposeful simplicity to enable a *deployment architect* to replace paper data collection with mobile data collection meant the tool suite lacked features required for certain use cases.

To evaluate the effectiveness of ODK 1 in meeting the data collection needs of global development organizations, we began our first phase of ODK 2 research with a survey of 73 organizations; we received responses about 55 different deployments from over 30 countries [22]. From the survey responses and our observations, we identified four principal areas of improvements for ODK 1:

1. *support data aggregation, cleansing, and analysis/visualization functions directly on the mobile device by allowing users to view and edit collected data;*
2. *increase the ability to change the presentation of the applications and data so that the app can be easily specialized to different situations without requiring recompilation;*
3. *expand the types of information that can be collected from sensing devices, while maintaining usability by non-IT professionals; and*
4. *incorporate cheaper technologies such as paper and SMS into the data collection pipeline.*

The design of ODK 1 focused on collecting data in the form of surveys, uploading completed surveys into a database in the cloud, and enabling different forms of export of data for analysis and aggregation. It did not provide the functionality for getting that data back out to mobile clients for review and update. However, the feedback about ODK 1 showed that many users wanted to be able to view previously collected data (either from past data collection or from a server database) on the device and use it to specify which data to display (e.g., a patient's past blood pressure readings) or to steer survey logic (e.g., select follow-up questions based upon a patient's medical history). One user told us, *"[One limitation of ODK 1] is the lack of a local database on the device [that contains] previously collected information. For example, the last time I visited your household, there were 5 people living here. Are those 5 people still living here?"* [22]. One of the most common requests that we received from users was to make it possible to view and edit data on the device. For example, another user told us, *"We need a presentable way of viewing collected data on the device ... like if you have a roster and need to make decisions based on some earlier responses, you need to be able to view this data"* [22].

Rendering of the surveys in ODK 1 was accomplished using a variant of a W3C XForms standard defined by the OpenRosa 1.0 APIs [150]. Although XForms can specify input constraints (to provide some immediate error checking abilities), form navigation logic (branching based on previous answers), and multiple languages (for local customization), it does not describe the visual presentation of the prompts and data types. Numerous survey respondents experienced difficulties adapting their workflows to ODK's generic user interface and JavaRosa XForm constructs. Some survey respondents referenced their struggles with XForms specifically with comments such as: *1) "I think most of the limitations have to do with Android, XForms, or formhub. I'm having trouble thinking of an ODK specific limitation"; 2) "complex branching needs a lot of manual XForm editing"; 3) "lack of support for a number of functions used in the XPath specification. This makes it particularly difficult to implement a pseudo-random number generator within the XForm"; and 4) "the XForm was difficult to manage."* Another user told us, *"We struggled to understand xml and the XForm. While the XForm is fairly simple, the xml structure is confusing. Some of the advanced features require core knowledge of xml coding*" [22].

One application domain where organizations experienced difficulties leveraging ODK was large complex medical protocols. Some of the difficulties experienced with digitizing medical *workflows* were complex logic in workflows that could not be easily expressed in XForms, questions requiring multiple input values (e.g. blood pressure, pulse and oxygen levels), and access to patient history. Organizations struggled to adapt these workflows to XForms because it required custom modifications that were beyond the technical capacity of many organizations. Extending the navigation expression language or customizing a question widget requires changes to ODK *Collect*'s Java source code, recompiling libraries, and generating a new Android app. This large barrier to customization slowed ODK adoption in medical protocol applications because organizations lacked the skills or funding necessary to customize the tools to their deployment needs. The problem with these types of large medical workflows is that the complex branching and user-directed non-linear flow requirements are not easily expressed in JavaRosa XForm's relevancy model.

Originally, ODK assumed humans would enter data by either directly entering data into the mobile device, use built-in sensors (e.g., GPS) to collect data, or use the camera to capture barcodes, pictures, and video. It did not support the ability to interact with new external sensors or process data captured. However, obtaining data from external sensors was an often-requested feature; such requests ranged from enhancing a health survey with data obtained from medical sensors to automatically incorporating GPS and compass data with captured photos. One user told us, *"Our [use case] requires us to measure the height of trees. We currently use a clinometer for this and enter the data manually. It would be great if we could access the clinometer [from the device] and use it as part of our data collection process*" [22]. Collecting data from sensors attached to the mobile device is attractive because applications can directly receive and process the data, obviating the need for manual data transfer by a human, which may be error-prone.

Finally, many organizations have extremely limited financial resources and still rely on paper forms or very cheap mobile phones to gather data, thus there is a need to connect these media to the ODK ecosystem. In addition, many organizations are unable to afford the cost of purchasing and maintaining a mobile device for every field worker. Such organizations would prefer to use cheap and well-understood paper forms to collect data at the lowest level of the information hierarchy, and then digitize the data at a higher level to enable data transmission, statistical analysis, and aggregation.

The limitations of the original ODK 1 tool suite were addressed by the design of ODK 2, a refined and expanded toolkit with a more flexible system architecture. While many of the details have evolved, the four design principles that were outlined in the ODK 2 vision paper [22] for refinement and expansion of ODK remain the same:

- *when possible, user interface elements should be designed using a more widely understood runtime language instead of a compile-time language, thereby making it easier for individuals with limited programming experience to make customizations;*
- *the basic data structures should be easily expressible in a single row, and nested structures should be avoided when data is in display, transmission, or storage states;*

- *data should be stored in a database that can be shared across devices and can be easily extractable to a variety of common data formats; and*
- *new sensors, data input methods, and data types should be easy to incorporate into the data collection pipeline by individuals with limited technical experience.*

ODK 2 tool suite was launched to support the common scenario in which organizations use previously collected data to influence their next action when revisiting a specific location. Examples of such a scenario include logistics management, patient follow-up in medical care, and environmental monitoring. In these scenarios, users often return to locations and reference previously collected data that they either verify or update. Each visit may require field workers to complete multiple forms, produce multiple data records, and/or require multiple review steps. Each phase may draw upon data collected during any other phase or data found in supporting information tables; thus, this data needs to be available even when disconnected. Revisiting data from previous surveys is not supported in ODK 1.

Expanding on the four basic ODK 2 design principles, a larger list of features was generated based on specific use case requirements [20]. For example, dynamic value checking based on previous data was required to improve data integrity, which is seen as a key benefit of digital data collection. An agricultural longitudinal study is an example use case being able to use previous data to improve data quality. Generally, these studies have agricultural extension workers visit crops multiple times during a growing season. The workers track the progress of the crop over time so they can compare growing conditions to try to improve crop yields. However, the validation logic in ODK 1 uses static formulas so the values check can only use a static absolute min or max value. Unfortunately, the values for a reasonable min and max value are dynamic and differ between early in the season versus later in the season. Thus, to better support this type of longitudinal study, ODK 2 should allow its *data management applications* to access previous crop heights and use dynamic calculations to catch data anomalies. ODK 2 expands its scope from ODK 1's *data collection platform* to a *data management platform.* A mobile "data management application" (designated as an

'application' to differentiate from the previously defined 'app') is the customized set of activities created by an organization to perform its data collection and management workflows in order to accomplish its business objective. The following is an expanded list of the key design goals of the ODK 2 frameworks [20]:

- *Workflow navigation should use intuitive procedural constructs, function independent of data validation, and allow for user-directed navigation of the form.*
- *The presentation layer must be independent of the navigation and validation logic.*
- *The presentation layer must be customizable without recompiling the Android apps via HTML, JavaScript, and CSS.*
- *Partial validation of collected data should be possible and the validation logic should be able to be dynamic.*
- *Local storage should be robust and performant for data curation and for longitudinal survey workflows using a relational data model.*
- *Multiple data collection forms should be able to modify data within a single, shared, data table.*
- *Foreground and background sensors should be supported for data collection.*
- *Should support adding new sensing and other methods of input*
- *Disconnected operation should be assumed as data must be able to be collected, queried, and stored without a reliable Internet connection. When the Internet becomes available, the framework and cloud components should efficiently replicate data across all devices.*
- *User and group permissions are needed to limit data access.*
- *Cloud components should be able to fully configure the data management application remotely as well preserve a change log of all collected data.*

The multi-visit scenario requires bidirectional synchronization of data, support for complex workflows, references or updates to previously collected data, and security and data isolation for specific users. Working with partner organizations, we constructed a list of

common requirements that were not met by the ODK 1 tools. These requirements were developed both through initial design sessions as well as lessons learned from field deployments. Some key requirements for ODK 2 that were beyond the scope of ODK 1 are [20]:

- *Complex / Non-Linear Workflows*
- *Linking Longitudinal Data to Collected Data*
- *Data Security and User Permissions*
- *Reuse of Data Fields across Forms*
- *Bidirectional Synchronization*
- *Customizable Form Presentation*
- *Custom Apps Built with a Runtime Language (JavaScript)*
- *Sensor Integration*
- *Paper Digitization*
- *Custom Data Types that Update Multiple Fields in a Single User Action*

These new requirements and goals led to the development of several new tools (described in Section 3.4) that aimed to improve the Open Data Kit project by expanding the functionality available to organizations deploying information management solutions in low-resource regions.

### *3.1.1 Mezuri Data Pipeline*

As part of the process of evaluating and refining the Open Data Kit project, we started a collaborative project with the University of California, Berkeley and other universities. The project was named "Mezuri" which is Esperanto for "measure." The goal of the Mezuri project was to create a configurable generic data collection and storage platform that enabled domain experts to gather, process, and share data. Mezuri focused on building end-to-end *workflows* and *dataflows* to support monitoring and evaluation (M&E) activities by combining data from multiple sources (e.g., surveys, sensors, public data sets). As part of the Mezuri project the team published, in Kipf et al. [77], a list of requirements to support three key functionalities: data collection, data processing, and data analysis. Since ODK 2

was designed to be part of the Mezuri pipeline, Mezuri's requirements were part of the requirements gathering process for ODK 2. The Mezuri requirements [77] are shown in Table 3.1.

Table 3.1: Mezuri's derived engineering requirements from Kipf et al. [77]

| *Requirement* | *Description* |
|---|---|
| **Abstraction** | The system shall allow interchanging of components of the platform (e.g., the backing datastore) to address varying user needs (e.g., storing both survey and sensor data). Additionally, an abstraction layer shall allow authentication throughout the platform. |
| **Accuracy & Transparency** | All imported data and any subsequent corrections to that data are retained and held distinct from each other within the datastore (immutable). Each correction maintains provenance information distinct from that of the imported data (traceable). The system shall be capable of re-running a processing step upon a specific snapshot of a corrected data set and this shall produce the same results (deterministic). |
| **Common schema** | The schema of the data and the interfaces of the processing may be standardized to allow automatic determination and recommendation of processing tools. |
| **Durability** | The storage service shall be designed in a way that prevents any data loss when any single process of the system fails. Similarly, the processing of data shall support fail-overs to avoid recalculating entire workflows. |
| **Isolation** | Processes in the platform shall be isolated from each other to avoid side effects. Further, any user-defined code shall be isolated in its own runtime to protect the platform from malicious code. |
| **Privacy** | Often, data collected is sensitive and controlled by institutional review boards. The system shall be compatible with privacy, security and anonymization requirements of most IRBs. |
| **Provenance** | The platform shall track transformations performed on the data, including user modifications, to be auditable and to support repeatable workflows. At the same time, external provenance metadata shall be imported into the platform to have a complete trace of the provenance chain. |
| **Revision of Provenance Metadata** | Provenance metadata shall be immutable by default, however, updates to sensor configuration may occur after the data was imported into our platform. For auditing purposes, Mezuri shall track the history of any update. |
| **Scalability** | As the volume of data increases and the number of users grows, the platform needs to be able to scale to meet the additional data and processing requirements. |
| **Sharing** | Mezuri shall allow researchers to share their data, processing tools, and workflows within and outside of the platform. When sharing their tools within the platform, users should be able to share them as black boxes that allow others to process their data without having access to the actual (sensitive) code. Shared workflows should include initial input parameters and input data sets. Workflow owners should be able to control at which granularity level someone can access the workflow. |

### *3.2 Challenges of existing mobile application paradigms*

Building and deploying mobile data management and decision support applications can be challenging, particularly because the task of bridging the design-reality gap [65] is left to the non-programmer *deployment architect*. Experiences from the field highlight the fact that not all the collected data is considered equally valuable. Therefore, when software frameworks select the method of data transmission, they should take into account the properties of the data being transmitted. This section discusses our investigation into existing synchronization options and outlines some of the design assumptions that inhibit deployment customization across diverse networking environments. New software frameworks designed to include the multi-perspective design with additional roles (as described in Section 1.1) are needed to resolve the challenges with deployments in resource-constrained environments. The challenges in resource-constrained environments often magnify problematic design paradigms because of the different design constraints that exists in different parts of the world.

**Uniform Data:** Existing routing paradigms often assume that *inherent data characteristics* (e.g., data types, data size) are sufficient to determine the appropriate network technology for data transmission [38, 126]. This assumption overlooks the fact that data is not uniform but instead has two distinct types of characteristics: *inherent* and *contextual*. As discussed in Section 1.1, data is often transmitted using whatever protocol a software developer selected when creating the software. Dynamic selection of available protocols based on a deployment's context and user location could improve connectivity in challenged network environments. To better facilitate dynamic selection the traditional concept of the TCP/IP Application Layer [132] should be extended to include: 1) metadata from the platform about connectivity; 2) data properties from the software/application developer; and 3) contextual constraints from an *deployment architect*. The selection of data transmission method should involve accounting for *inherent data characteristics*, *contextual data characteristics*, and network properties [23]. The *inherent data characteristics* are independent of the application domain, while the *contextual data characteristics* (e.g., data priority, data

importance, deadlines, and precedence) are necessarily dependent on the application usage scenario and other contextual information. Examples of other *contextual data characteristics* include an organization's data policy and local laws that can affect how data is allowed to be stored and transmitted (e.g., private medical records vs. public data sets).

**Single-Task Mobile Apps:** Resource-constrained environments often lack sufficient technical personnel to build and customize information systems. This leads organizations to use productivity software (e.g., MS Excel, MS Word) to create solutions that can be customized by staff having little programming expertise. These software tools were designed to be used on conventional personal computers (PCs). Unfortunately, conventional PCs are often poorly suited to be used in limited infrastructure environments. Although mobile devices are well-suited to scarce connectivity and sporadic grid power, mobile software tools do not yet offer the same range of features as PC productivity software. Instead, several small apps focused on single tasks are created, leading to specialized apps with minimal customizability. Additionally, with single-task apps there is often limited coordination of system resources, making it challenging to conserve resources. For example, multiple apps could simultaneously attempt to communicate when connectivity becomes available. Mobile frameworks, such as ODK, are needed to help organizations customize and refine their apps to their context while maintaining the single-task paradigm.

**Similar Transmission Cost:** Developers often choose a single transport protocol such as TCP/IP or SMS because of systems abstractions and availability of networks at the original deployment location. However, the cost associated with connectivity can vary across different regions, which creates feasibility issues for deploying applications in varying contexts. For example, a 500 megabytes post-paid mobile broadband subscription in Europe costs 1% of per capita of gross national income (GNI). By contrast, the same subscription costs 38% of the average per capita GNI across Africa [139]. Even as the cost of broadband subscriptions falls globally, an entry-level broadband connection continues to cost over 100% of per capita GNI in less developed countries, as compared to only 1% of per capita GNI in more developed countries [138]. Even in developed regions, there are communities yet to be

covered. Although technologies with universal connectivity options like satellite uplinks exist, financially constrained organizations cannot afford them. Restricting data transmission to a single protocol can lead to missed opportunities in optimizing transmission costs based on contextual qualities of data.

### *3.2.1 Existing Synchronization Tools Measurement*

Commercial cloud technologies, such as Dropbox [46], Google Drive [57], and OneDrive [92], have simplified synchronizing files between devices. We evaluated the three cloud solutions [23] and found they lack the support needed to enable organizations to treat data differently based on contextual data qualities and do not allow for alternative connectivity options in challenged network environments. Additionally, problems with the file synchronization approach were reported organizations during field interviews. Organizations would find that their spreadsheets would constantly get into a conflict state because of multiple devices operating in intermittent connectivity being unable to constantly send and receives updates. This is one of several reasons ODK 2 uses a database row as a smaller unit of change. This fine-grain data synchronization enables field workers to go about their daily tasks recording the small portions of data that are changing instead of trying to maintain a "global" correct state of the data in challenged network environments. Database synchronization on mobile devices has received less attention because of the general availability of the cloud for data storage in resource-rich contexts.

To better understand the performance of existing tools, we the measured performance of Dropbox, OneDrive, and Google Drive on Android devices using libraries provided by the cloud services. Since not all contexts have similar networking characteristics, we chose three cities (Lima, Peru; Kisumu, Kenya; and Lahore, Pakistan) from different continents that were not located in a 'high-income' country. Large cities generally have better infrastructure than rural areas, so this comparison illustrates best-case scenarios for countries with resource constraints. To provide context for the performance divide between urban and rural areas, we also evaluated the round-trip time (RTT) differences of an urban city in the U.S. (Seat-

tle, WA) and rural towns in the U.S. (Chowchilla and Pala, CA), which have populations of ˜600K, ˜18K, and ˜1.5K respectively. It should be noted that the values measured in Chowchilla and Pala represent a best case for rural connectivity, as our measurements were taken from the more densely populated town centers. Service carriers used in the experiments include Claro (Lima), Telenor (Lahore), T-Mobile (Seattle), Verizon (Chowchilla), and AT&T (Pala). File synchronization was measured 15 times per service using 10 kilobytes, 500 kilobytes, 1 megabyte, 2 megabytes, and 10 megabytes of image files and 1 kilobyte, 10 kilobytes, 100 kilobytes, and 1 megabyte of text files.

Table 3.2: Network measurements from various locations

| LOCATION | RTT (ms) | BANDWIDTH (Mbps) | LOSS (%) |
|---|---|---|---|
| Lima | 1173.0 | 1.05 | 0 |
| Lahore | 207.0 | 1.05 | 0 |
| Chowchilla | 154.6 | 1.69 | 0 |
| Pala | 63.0 | 6.93 | 0 |
| Seattle | 39.2 | 11.00 | 0 |

Table 3.2 shows network statistics recorded using ping and 'iperf' tools. Cellular networks in Lima and Lahore had higher latency with low bandwidth capacity, with Lima experiencing the longest round trip time at 1,173 ms. Seattle was 5 and 30 times faster than Lahore and Lima respectively. Mobile network connectivity in rural US test sites was significantly lower than the urban test site, as Chowchilla had 1.69 Mbps (154.6ms RTT) and Pala had 6.93 Mbps (63.02ms RTT) compared to Seattle's 11 Mbps (39.2ms RTT) high-speed bandwidth. Poor network infrastructure is not just a problem for low-resource countries, but also for rural parts of high-resource countries as well.

The results from synchronization performance tests of the three popular cloud-based systems are shown in Figures 3.1 through 3.3. The figures show that Dropbox (Figure 3.1) performed better than Google Drive (Figure 3.2) or OneDrive (Figure 3.3), with lower synchronization times for all tested file sizes. This is unsurprising as the Dropbox API compressed all files prior to transmission, while Google Drive and OneDrive did not. OneDrive only performed differential synchronization of Microsoft Office files, which excludes many large media files. Google Drive did not use differential synchronization for any file, causing it to use the most bandwidth per file update of the three evaluated options. When accessed

via the developer API, Dropbox did not provide differential synchronization, though it did perform data compression prior to transmission. We also noted that OneDrive experienced higher variability in file transfer times than Google Drive and Dropbox, with a standard deviation of 5.2 seconds compared to 2.3 seconds for Dropbox and 4.6 seconds for Google Drive. Even though Lahore, Kisumu, and Lima have access to mobile Internet, the performance was inconsistent and more prone to high latency and limited bandwidth than in Seattle.

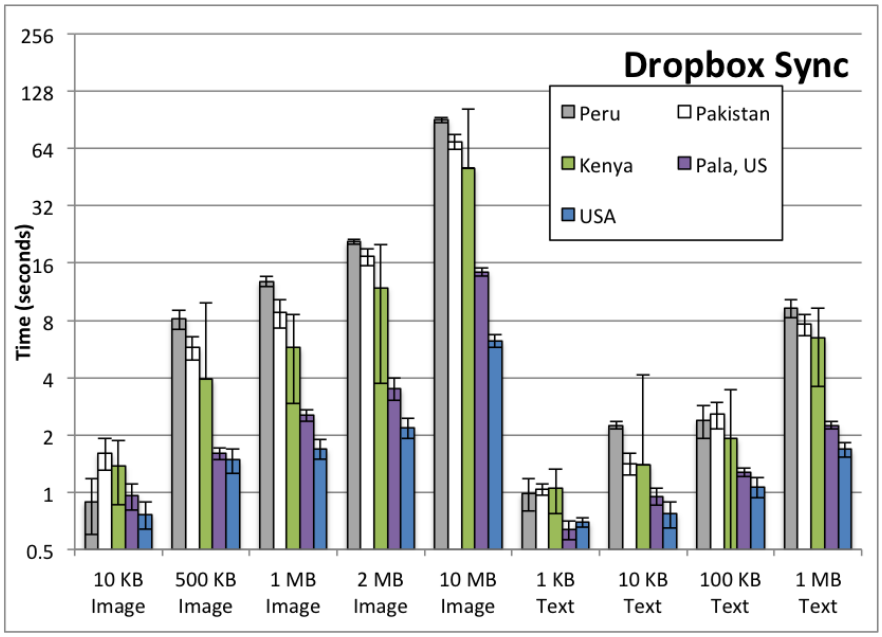



Figure 3.1: Dropbox file synchronization performance with varying file sizes using mobile data connection. (Log Scale) [23]

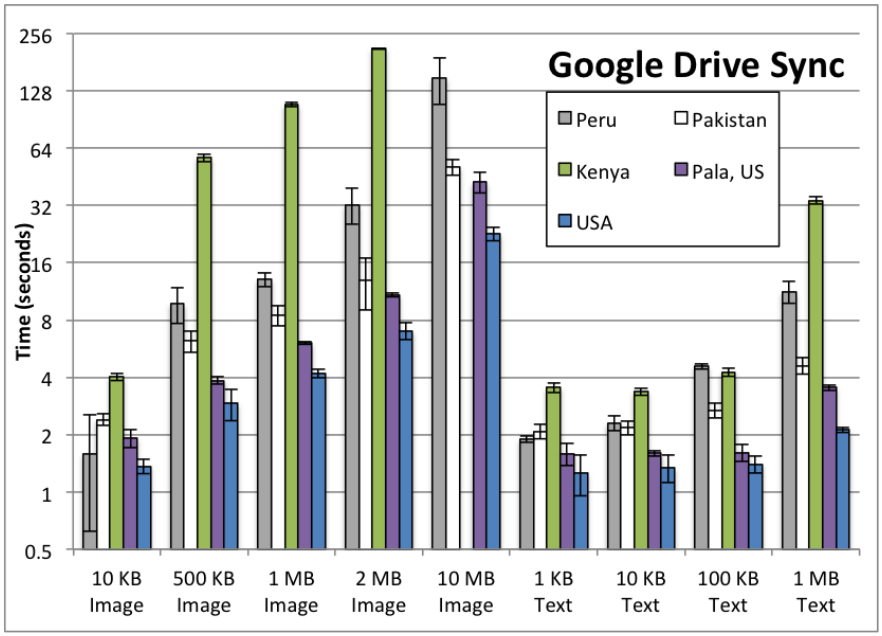



Figure 3.2: Google Drive file synchronization performance with varying file sizes using mobile data connection. (Log Scale) [23]

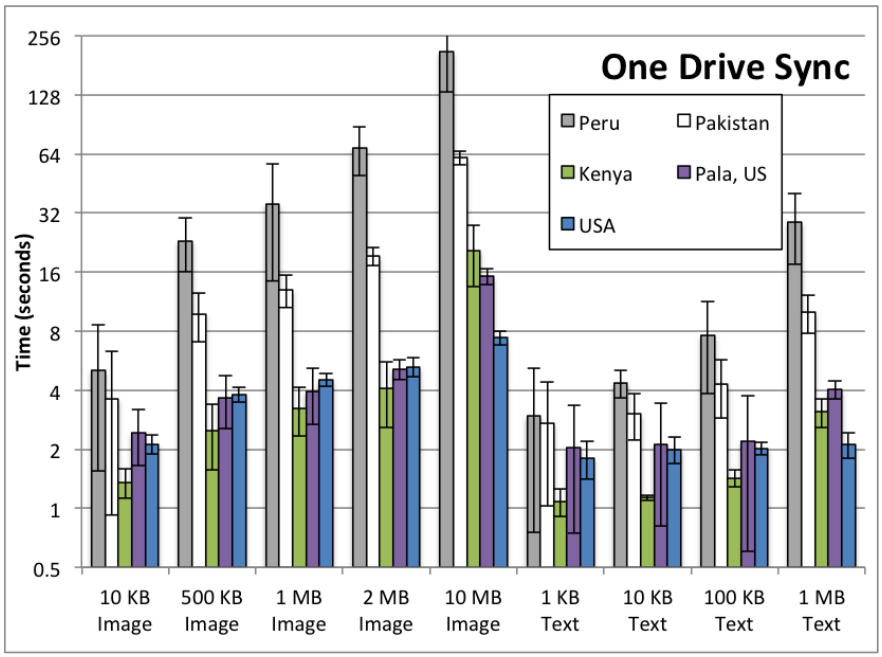



Figure 3.3: OneDrive file synchronization performance with varying file sizes using mobile data connection. (Log Scale) [23]

Our measurements demonstrate that common data synchronization platforms experience issues in varying network environments, as they all experienced longer file transfer times and greater variability in transfer time in resource-constrained environments. Based on our experiments, Dropbox would be the preferable 'off-the-shelf' solution, followed by Google Drive, then OneDrive. However, there are still issues that these cloud synchronization platforms do not address such as: 1) they lack support to enable organizations to treat data differently based on contextual data qualities; and 2) they are TCP/IP based and do not allow for alternative connectivity options in challenged network environments.

### *3.3 Related Work*

A variety of existing solutions and application frameworks attempt to solve some of the issues outlined in this chapter. What distinguishes ODK 2 from other solutions is its focus on providing a suite of interoperable tools that work together to provide base functionality for flexible information management systems that can operate in low-resource contexts. For example, Apache Cordova [4] (the open-source version of Adobe's PhoneGap discussed in Section 2.3) and ODK have different overall focuses. Cordova is designed for programmers to build mobile applications and helps solve the problem of cross-compatibility for different operating systems. Similar to ODK 2, Cordova uses a HTML/JavaScript and a custom native SQLite library; however, unlike ODK 2, Cordova targets programmers. ODK 2 focuses on making it simpler for non-programmers to create mobile applications on the Android devices and enable organizations to conform to their low-resource constraints.

Many research projects (e.g., Bayou [135], Coda [78], PRACTI [11]) enable disconnected operation for mobile computing through client-server replication systems. However, many of these systems are not necessarily designed to be adaptable to workflows of global development organizations that share subsets of data and manage data access restrictions for large numbers of users with limited technical staff. Several research projects have tried to simplify mobile development by providing abstracted table programming interfaces that assist mobile app programmers with data management. Two examples of such projects that

build database-table-like schematics are Izzy [61] and Mobius [30]. Like ODK, Izzy tries to make application development easier by providing a simple programming interface to access database tables; however, the target user is a programmer, making it difficult for low-resource organizations to use. Mobius [30] creates a unifying and data serving abstraction for mobile apps that provides automated caching, predicates, notifications, and protocols for disconnected operation. Interestingly, Mobius presents the programming abstraction of a logical table of data that spans devices and clouds. However, Mobius does not match humanitarian organizations' requirements as those organizations do not need the database predicates or notifications because of their need to minimize the amount of network traffic to keep operating costs low and to synchronize only when requested by the user. This approach becomes problematic for long periods of disconnected operation, which are expected to be experienced by ODK users.

Computing for global development research has produced multiple mobile application framework research projects. As discussed in Section 2.3, CAM [104] was one of the earliest mobile application frameworks for resource-constrained environments but it was tied to limited number of J2ME phones and the custom scripting language was seen as a barrier to entry as it required a new skill that someone had to acquire. In comparison, ODK 2 exposes JavaScript as its customization language to allow the tools to have a pre-existing base of trained programmers. Another early example was the e-IMCI project [42], which used personal data assistants to encode complex medical workflows. While e-IMCI was not customizable, it provided an early example of how mobile computing platforms could improve an organization's workflow in remote locations. DeRenzi et al. went on to investigate how to improve health care worker performance by closing the feedback loop through voice-based or web-based information [43, 41]. Instead of requiring additional infrastructure to communicate information back to field workers, ODK 2's bidirectional synchronization enables organizations to build this communication into their workflows. DeRenzi's et al.'s observations about information flowing back to the field worker match anecdotal field conversations about how field workers do not necessarily understand the benefits of the digitizing work-

flows, as they often do not see the difference of sending paper into the office versus having it transmitted. The Uju project [148] focused on making it easier to create database-centric applications for SMS, but, unlike ODK 2, it was not designed to integrate with multiple tools that obtain data from various inputs (e.g., sensors, paper forms). What distinguishes ODK 2 is its focus on providing a suite of interoperable tools that provide the necessary functionality to build flexible information systems.

The CommCare framework with CaseXML [39, 40] began with similar design goals to ODK, as both originally targeted rural healthcare workers and used the same XForm standard. CommCare enables organizations to customize their field workers' digital workflows using XForms and enables workers to share 'cases' through their CaseXML. CommCare operates on both J2ME and Android devices. ODK 1's *Collect* software was the base software for CommCare's Android app that was expanded by adding features to work with CommCare's CaseXML. While CommCare has been a successful platform for organizations with field healthcare workers, other organizations reported problems adapting CaseXML to other domains. The limitations reported by organizations with adapting their workflows to the CaseXML model contributed to the decision to abandon XForms entirely in the ODK 2 design.

Similar to ODK's disconnected operation philosophy, Apache's CouchDB [5] has an "offline-first" philosophy that includes a replication protocol that maintains a copy of the data. However, CouchDB is a JSON document-based database instead of a table based database with rows of data. Unfortunately a document-based database does not cleanly map to the schema-enforced model of data management in ODK 2. ODK 2 uses a tabular data format with schema to help push deployment architects to understand their data collection needs and think through their deployments. We have observed organizations using document databases to collect many different types of data that go unused because they are unable to resolve conflicts and merge data because of inconsistently stored data (possibly differently formatted). Furthermore, unlike the simple process of importing CSVs of tabular data into Excel, the commonly used MapReduce paradigm used for document-based

databases often requires programming skills.

Tablecast [134] is an XML protocol that extends the Atom syndication format [58] to handle concurrent modifications to tables of data. Tablecast relies on Atom's publish/subscribe architecture where each node publishes modifications to a table as a feed, and multiple feeds from multiple publishers are merged together to determine the latest state of a table. With mobile devices' limited battery life and the varying network connectivity conditions that can exist in low-resource environments, the Tablecast architecture is problematic to meet the requirements of the synchronization protocol because many mobile devices could be unreliable as feed providers. One of the ODK design requirements is to optimize for disconnected operation.

Oracle's Database Mobile Server [100] fulfills many data storage and synchronization requirements of ODK 2, but it is a proprietary solution making it challenging to customize and extend. The licensing fees to Oracle's database are also expensive, causing it to be a problematic solution for financially challenged humanitarian organizations working in low-resource contexts. Some of the key features that the Oracle database provides are synchronization of tabular data across mobile phones, sophisticated user access control, conflict resolution, and adding arbitrary processing logic into the *dataflow*.

## *3.4 ODK Tool Suite 2*

The ODK 2 tool suite is a parallel effort (not a replacement) to the ODK 1 tool suite; it provides the general-purpose tools for a virtually unlimited set of use cases and enables organizations in resource-constrained environments to build, deploy, maintain, and ultimately own mobile data management applications and business logic. ODK 2 is an open-source tool suite aimed at enabling organizations to easily build application-specific information services for disconnected environments. This section gives a brief introduction of the ODK 2 tool suite (individual tool names are italicized) with additional details about frameworks in Chapter 4.

ODK 2 seeks to satisfy an organization's particular requirements through the composi-

tion of narrowly focused tools that provide multiple customizable frameworks rather than through a single monolithic app. Creating a single app that tries to provide abstractions for every usage scenario could have overwhelming complexity. ODK 1 only had a single Android app (*Collect*) that provided a question-rendering framework for XForms. With ODK 2 we wanted to expand the functionality but keep the system design modular. To simplify customization we developed multiple reusable frameworks that are designed to smoothly transition between frameworks to give the feel of a unified application. The collection of ODK 2 frameworks provide a set of diverse features that enable organizations to create customized data management applications on Android devices. Key aspects of the architecture are:

1. a **modular design** that enables individual software frameworks designed for particular tasks to be used together to achieve complex workflows,
2. **data and configuration management** that enables data protection, storage, and sharing of data across mobile devices and cloud components,
3. a **synchronization protocol** that is designed to operate in challenged networking conditions, and
4. a **services-based architecture** that abstracts common functionalities behind a consolidated unifying services layer.

The evolved ODK 2 system architecture is still governed by Open Data Kit's over-arching concern that for computing tools to address the many information gaps in low-resource regions, information services must be composable by non-programmers and be deployable by resource-constrained organizations (in terms of both financial and technical resource constraints), using primarily consumer services and devices. ODK 2's design and development was guided by the few simple Open Data Kit design principles discussed in Section 2.1.

When designing ODK 2, we focused on creating frameworks that have multiple abstractions that can perform the same task to help users with different skill levels configure their system with varying options. These frameworks are described in further detail in Chapter 4.

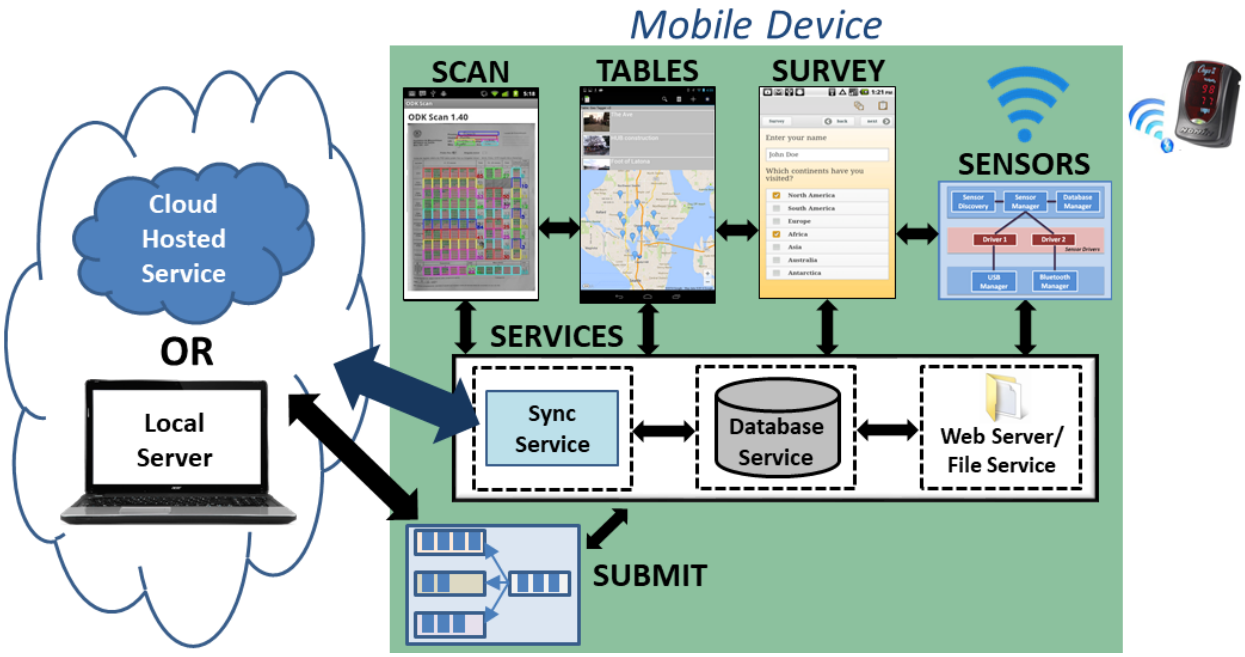



Figure 3.4: This figure shows how the ODK 2 frameworks can be used in concert to create customized mobile data management applications [20]. The six ODK 2 mobile framework 'apps' that comprise the ODK 2 tool suite are *Services*, *Survey*, *Tables*, *Scan*, *Sensors*, and *Submit*.

Instead of a one-size-fits-all design for the interfaces, ODK 2 creates multiple interfaces based on the amount of customization that is needed (which can sometimes correlate to skill level). These multiple interfaces target different skill levels in an attempt to emulate the concept of design scaffolding (as exemplified in [113, 3]) to enable users to learn more about the ODK 2 frameworks through the use of simple interfaces that are less confusing/intimidating. ODK 2 is a flexible, open-source platform that enables organizations to empower field workers to collect, manipulate, and view data on their Android devices, share data between devices, and synchronize data to the cloud. To expand ODK's ability to handle new features and complex workflows, ODK 2 abandons the XForm standard and creates a more flexible questionnaire-rendering tool called *Survey* to complement ODK 1's *Collect* tool.

The six Android apps shown in Figure 3.4 each contribute a framework to ODK 2. Each framework contributes abstractions to specific areas: *Tables*, the data curation framework;

*Survey*, the question-rendering and constraint-verification framework; *Sensors*, the device connection framework; *Services*, the common data services framework; *Scan*, the paper digitization framework; and *Submit* the communication framework. Each framework provides specific functionality into reusable abstractions for ODK 2. To fulfill complex requirements, the deployment architect uses the necessary subset of frameworks in concert to build a customized data management application. The diversity of global development requirements necessitates a modular framework that:

- Enables organizations to leverage mobile devices to build customizable information systems.
- Isolates the user-configurable portion of the framework from the reusable system components to create abstractions that are flexible and adaptable enough to support many different types of workflows from different subject domains.
- Facilitates the integration of new capabilities (e.g., sensors, improved network performance algorithms, viewing data) into the framework with abstractions that enable new components to be added. Integrating new sources of information (e.g., surveys, sensors) should be as simple as adding configuration files and data-handling routines to the framework.

ODK 2 abstractions are based on this multi-perspective analysis that considers various actors' roles and skills that contribute to deploying a mobile data management solution customized to the organizations' requirements. As discussed in Section 1.1, there are three basic perspectives: the 'platform perspective,' the 'application developer perspective,' and the 'application deployment perspective' [23]. The 'platform perspective' incorporates the device and operating system perspective on aspects such as connectivity, power, features, and mobility. The 'application developer perspective' incorporates the perspective of the software developer, who chooses what functionality and features the ODK platform exposes. The 'application deployment perspective' encompasses the dynamic issues relating to contextual deployment requirements such as workflows, information prioritization, and financial

restrictions that are unknown when the ODK source code is compiled. Unfortunately, developers likely do not fully understand how future users will want to deploy the framework in different contexts with varying connectivity, budgets, laws, data policies, user education level, etc. Therefore, ODK focuses on creating frameworks that give flexibility to the *deployment architect* who has an 'application deployment perspective' and understands the contextual deployment requirements that are often dynamic. Unlike standard reusable model-view-controllers, ODK 2 attempts to create abstractions usable by *deployment architects* by emulating the design of PC productivity software that targets non-software developers. A *deployment architect* should be able to use ODK 2's XLSX specification to create a complex *workflow*. An *end-user* should be able to use the system to solve impromptu issues during the field deployment while being protected (with user permissions) from misusing data or doing something wrong. Chapter 5 discusses the experiences of *deployment architects* and *end-users* using and configuring ODK 2 frameworks.

### *3.4.1 Data Management on Mobile Device*

Many applications rely on previously collected data; for example, logistics management, public health, and environmental monitoring often require workers to return and reference previously-collected data to verify and possibly update conditions. In the ODK 1 design, revising data from previously completed surveys was not supported. However, many ODK users wanted to be able to use all or part of previous surveys to complete new surveys (e.g., not re-entering patient demographics for a follow-up visit when that data was already collected in the original registration form). To enable data updating, aggregation, curation, cleaning, and analysis functionality on the mobile device, we created the *Tables* framework. *Tables* presents a user interface for editing and viewing tabular data that is optimized for the smaller screens of mobile devices [21]. *Tables* enables customizable views defined by HTML/JavaScript files, thus making the user interface much more flexible without the need to compile a new Android app. These views can pull data from, and link to, other tables, enabling *deployment architects* to form an integrated application workflow. For example, a

table of facilities can link to a table of specifics about individual resources at each facility. Alternatively, *Tables* can use *Survey's* strong data typing to add and edit entire rows and incorporate input constraint checking, or use *Scan's* image processing capabilities to add rows based upon filled-in paper forms.

User experiences from ODK 1 deployments [62, 22] demonstrated that non-technical users were able to make small customizations to existing XForms, but creating an entire XForm from scratch was often too challenging because of the syntax. *Build* was created to simplify XForm creation in ODK 1 to shield users from the complexity of writing XForms as *Build* allows a user to graphically compose questionnaire. However, some users found the graphic interface too cumbersome for creating complex XForms and for sharing an XForm design. Users wanted to be able to share their XForms to enable another user to easily reuse portions of previous questionnaires. To simply form development, Columbia University created 'pyxforms0' (later named ODK *XLSForm*) to give users the option of writing their questionnaires in an Excel spreadsheet that could automatically be converted into an XForm. Based on the success and popularity of *XLSForm* tool, we decided to leverage Excel spreadsheet as a basic configuration file for ODK 2. Since ODK 2 was not following the XForm standard we built a revised converter to transform a spreadsheet to a JSON description that could be rendered using ODK 2's new frameworks that leveraged web technologies. By allowing users to specify information in a spreadsheet, it enables non-technical users to remain shielded from the complexity of writing JSON and JavaScript. Users with minimal JavaScript and HTML skills were able to copy and modify standard template files (e.g., use different HTML constructs or add CSS style classes) and reference these modified template files to customize the rendering of individual questions in the form or create new question types. In the same way, users can also revise the standard templates and CSS style sheets to create an organization-specific look and feel. Users with more advanced JavaScript and HTML skills can customize a question widget's event handling (e.g., add mouseover-like treatments) or define entirely new widgets.

ODK 2 data management applications consist of survey definitions, web content, config-

uration files, data tables, and data rows. These components are consumed by the framework to define the control flow, user interface, schema, data validation logic, and business logic of the application. Departing from the ODK 1 design, which uses an XForm-based hierarchical document model, the ODK 2 frameworks store data in a relational database on the Android device. The presentation layer can execute arbitrarily complex SQL queries (e.g., joins), vastly increasing the expressive capability of the ODK 2 frameworks. This expressiveness is required for complex workflow decisions, such as the disaster response case study in Section 3.5, that requires queries of previously collected beneficiary data to determine future distributions. A relational database also permitted more complex relationships to be established between data sets. Dependent data sets, similar to ODK 1's 'repeat groups' concept, that were previously stored in the same XForm document instance, are instead stored in a separate data table allowing the *deployment architect* to define business logic to maintain the linkage of these data sets with other data sets.

ODK 2 uses CSV files as an interchange format to easily move data between ODK 2 and other systems as CSV is a common format that is used to encode data that can be easily imported into databases, Excel workbooks, statistics programs, or other software programs. Deployment architects are easily able to import existing data into ODK 2's database using CSVs. The ODK 2 database uses several tables to manage an application's metadata: one for table properties (table_definitions – an entry for each table) and one for column properties (column_definitions – an entry for each column in every table). The table definitions table contains information about the user-created tables, and has columns for data such as the table ID and the table's name. Another table acts as a key-value repository to store the settings required to correctly display the table. This includes such data as display name, column order, and font size for the default spreadsheet-like view. The column definitions table holds information about all the columns of each table, including the ID of the table of which the column is a member, the column ID, and the column name. Each column can have a data type (e.g., number or date range), which is used to restrict the values that can be put in the column. If a column's values are restricted to a limited number of specific

strings (such as the multiple-choice questions found on many paper forms), ODK 2 has a multiple-choice data type that includes a list of allowable data values for that column.

With ODK 2's design incorporating tables and rows of data (as opposed to the XForm document format of ODK 1), it makes it simple to use CSV files as an interchange format to move data between any combination of servers and clients. To assist organizations with moving and backing up data, the ODK 2 tool suite provides a tool called *Suitcase* that uses CSV files to update and backup ODK 2's servers. Additionally, *Tables* comes with built-in functionality to integrate with CSVs. If a user has data to import (such as from an existing Excel spreadsheet), they can put a CSV file on the device and use *Tables's* CSV importer to create a new table with data from the CSV file. Users can export CSVs from *Tables* and then can re-import it to a new device to create a table with columns, data, and settings from the original table.

### *3.4.2 Improved Input Methods*

ODK 2 reduces the amount of manual data transcription into surveys by making it possible to attach external sensors to mobile devices. By hiding complexities such as the management of communication channels and sensor state, as well as data buffering and threading, the *Sensors* framework [19] simplifies the code needed to access a sensor. *Sensors* provides a common interface to access both built-in and external sensors connected over a variety of communication channels. Users who want to integrate external sensors with their mobile devices download and install the *Sensors* framework and sensor driver app from an app store such as Google Play. This facilitates the easy delivery of the application and driver updates to devices. Sensors provides abstractions that delineate application code from code that implements drivers for sensor-specific data processing. The sensor driver abstraction allows device drivers to be implemented in user-space so that locked devices can be customized by end-users. The framework handles the data buffers and connection state for each sensor, which simplifies the drivers. Separating application code from the device driver code also allows the code bases to evolve independently.

In addition to adding sensor data, the continued use of paper forms for data collection in resource-constrained environments made it important that ODK 2 also facilitate efficient data entry from paper forms. Many of the paper forms used by organizations for data collection contain a mixture of data types, including handwritten text, numbers, checkboxes, and multiple choice answers. While some of these data types, such as handwritten text, require a person to manually transcribe the data, others, like checkboxes or bubbles, can be analyzed and interpreted automatically. To take advantage of machine-readability, other researchers designed the *Scan* framework [36], which facilitates the processing of existing paper forms. All of the image processing is performed on the device so as not to require an Internet or cellular connection. After the image processing is completed, *Scan* launches *Survey* so that users can manually complete the entry of data types that are not machine-readable. *Scan* makes this data entry process faster by exporting small image snippets of each form field in *Survey*, and the image snippets are displayed on the screen of the device alongside the corresponding data entry box so that users can simply look at the image snippet and type in the value displayed.

### *3.4.3 Data Management in the Cloud*

ODK 2 uses a cloud component to provide robust permanent storage of data and configuration information. The data management application interacts with the cloud component through RESTful interfaces. The ODK 2 cloud component manages the 1) user permissions (user authentication and group membership), 2) infrequently changing configurations (configuration files, form definitions, data table definitions), and 3) frequently changing data (content of the data tables and associated row-level media attachments). The configuration files enable the *deployment architect* to specify the *end-user's workflow*. Generally, once the *workflow* is established for an application, the *workflow* will change infrequently as compared to the data that workers collect and update, which is why the *workflows* are defined in files and the data is stored in a database. To support disconnected operations, the *Services* framework implements the ODK 2 synchronization protocol to maintain a snapshot

of the data onto each mobile device. Upon connecting to the Internet, the mobile device initiates the synchronization protocol to reconcile changes and update its local state to a new, reconciled snapshot of the cloud component's content. On the device, a user's identity is established via a successful login to an ODK cloud component. Thereafter, the user's identity and permissions are cached until the user resets his or her credentials or until the device is next synchronized with the cloud component. There are four classes of users: (1) anonymous or unauthenticated users, (2) authenticated unprivileged users with permissions to only read and modify a subset of the data, (3) authenticated superusers with permissions to read and modify all data, (4) authenticated administrators with permissions to read and modify all data, update user permissions, and change the configuration files that specify the data management application. Additional details of user permissions are discussed in Section 4.3.3.

In ODK 2, a simple row is the basic storage element; repeating groups are explicitly represented as linked rows across two different forms. This new design eliminates the complex backend mapping used in ODK 1 that made it difficult for organizations to access the database structures directly. The communications flow has changed so it is now a cloud-mediated, peer-to-peer store-and-forward network, enabling any authorized device to share data with any of its peers. Having a cloud service that acts as both the datastore and a synchronization master copy enables robust peer-to-peer operations in intermittent and low-connectivity environments. The cloud also provides a central point from which to manage and disseminate a security model that can be applied and enforced independently on each device. Data synchronization is currently initiated by the user because connectivity is often intermittent and organizations want to control data transfer costs.

ODK 2 Cloud-Endpoints are servers that communicate with *Services*. A Cloud-Endpoint is defined as any server that implements the ODK 2 REST Protocol. For our initial implementation, we built the data synchronization cloud service into ODK 1's *Aggregate* to leverage the security and database-agnostic storage layer, which enables operation from the cloud or a private server. *Aggregate* hosts the master table and is used to synchronize multiple

instances of *Services* running on client devices. *Sync-Endpoint* is a second implementation of a Cloud-Endpoint that is more modular and leverages Docker [45]. For ease of deployment and upgrades, *Sync-Endpoint* uses a micro-service design utilizing Docker containers. *Sync-Endpoint* does not store user information in its own database; instead, it integrates with several authentication services such as LDAP directory and Active Directory.

## *3.5 Case Studies*

Designing mobile data management systems for challenged network environments necessitates creating abstractions for deployment architects to configure and customize to their deployment context. To enable broad use by global humanitarian organizations, ODK 2 frameworks need to be designed without a priori knowledge of the application domain, deployment conditions, or how data from the various frameworks will interact. This means that the configuration abstractions need to be flexible enough to handle various field conditions but simple enough for a non-developer to make adjustments, as field conditions can and will change. To verify the accuracy of the requirements discussed in Section 3.1, we engaged field partners to validate the suitability of the ODK 2 frameworks to different use cases in resource-constrained environments. The validation of the requirements by a diverse group of organizations working on global health interventions, disaster response management, and other areas demonstrated the reusability, flexibility, and extensibility of ODK 2's frameworks[20]. Table 3.3 lists the required features needed by each of the case studies[20].

### *3.5.1 Childhood Pneumonia*

PATH, a non-government global health organization that supports health workers in low-resource environments, sought to create a mobile application, called 'mPnumonia', to improve healthcare providers' diagnosis and management of childhood pneumonia. Childhood pneumonia is a leading cause of disease deaths for children under five [54]. Previous projects had found that digitizing the World Health Organization's Integrated Management of Childhood Illness (IMCI) protocol on PDAs led to increased adherence while maintaining examination

Table 3.3: ODK 2 Case Study Feature Requirement Summary [20]

| | *Childhood Pneumonia* | *Chimpanzee Behavior Tracking* | *HIV Clinical Trial* | *Disaster Response* | *Mosquito Infection Tracking* | *Tuberculosis Patient Records* |
|---|---|---|---|---|---|---|
| Complex / Non-Linear Workflows | X | X | X | X | X | |
| Link Longitudinal Data To Collected Data | X | | X | X | X | X |
| Data Security and User Permissions | X | | X | X | X | X |
| Reuse of Data Fields Across Forms | | | X | X | | |
| Bidirectional Synchronization | X | | X | X | X | X |
| Customizable Form Presentation | X | | X | X | | |
| Custom JavaScript Apps | | X | X | X | X | X |
| Sensor Integration | X | | | | | |
| Paper Digitization | | | | | | X |
| Custom Data Types Update Multiple Fields in a Single User Action | X | X | | X | X | |

time [42]. However, when other organizations tried to re-implement IMCI using ODK 1, they struggled to map IMCI's complex medical *workflow* to XForms. ODK 2 addressed this complexity with *Survey*, which allowed a non-programmer global health intern to digitize the IMCI protocol using an Excel workflow description format. Furthermore, the layout, styling, and presentation of each prompt in the form was customized to meet the specific needs of the 'mPneumonia' project. These customizations were simplified by making changes to HTML and CSS, which enabled a rapid iterative design process.

The 'mPneumonia' project [54] added a patient's pulse and blood oxygen saturation reading to the IMCI protocol. ODK's *Sensors* framework connected the pulse-oximetry sensor to the mobile device, allowing sensor readings to be written directly to the patient record. ODK 1 only allows a single value to be recorded to the data model for an individual

user action. A pulse-ox sensor reads two values at a time, requiring updates to two database columns. ODK 2 allows the specification of custom data types with multiple database columns enabling the pulse-ox data to be recorded in a single action.

### *3.5.2 Chimpanzee Behavior Tracking*

The Jane Goodall Institute (JGI), an early adopter of ODK 1, successfully replaced multiple existing paper data forms with ODK 1. However, after several attempts, they were unable to digitize one of their most important data collection efforts. They needed an alternative to sequential question form-based data entry to be able to track chimpanzee behavior in real-time. Park rangers follow troupes of chimpanzees through the jungle and record behavior using a paper form. The paper form consists of a grid structure with names of chimpanzees and boxes used to indicate the chimpanzee's presence or absence and the proximity to the group leader. A new entry on the paper form is completed every 15 minutes by the park ranger. At the end of the day the information from paper form is transcribed into a digital database. Unfortunately, there was no way to enforce internal consistency or data quality when using paper, so JGI wanted to create a digital data collection solution. However, the paper form's grid structure prevented the conversion to an ODK 1 question-based *workflow*. Maintaining the format of the data collected was very significant to the organization, since it has been used for years and the rangers feel comfortable with it.

Conceptualizing the *workflow* of recording chimpanzees' behavior as a series of web pages enabled the paper form to be converted into an ODK 2 *Tables* application. An HTML table was used to create a grid that represented a database row that associated the current time and the chimp identification. Edits to the data HTML grid caused an update to the corresponding database row. Additionally, by *Tables* pre-populates chimp information from the previous time point, thereby lowering the data-entry task burden of the ranger by removing the need to re-enter data every 15 minutes. Instead, the ranger simply records changes because of the pre-population of data. While implementing this data collection *workflow* with *Tables* required some basic programming knowledge, it was less complicated than trying to

implement the same *workflow* using Android programming constructs. To implement the *workflow* in native Android would require knowledge of the Android persistence programming interface, familiarity with the Android user interface framework, and an understanding of Android's activity lifecycle. The *Tables*-based approach allowed developers to leverage ODK 2 frameworks for most functionality and to focus on the user interface to create a chimpanzee behavior tracking application.

### *3.5.3 HIV Clinical Trial*

Adaptive Strategies for Preventing and Treating Lapses of Retention in Care (AdaPT-R) was a multi-year University of California, San Francisco (UCSF) study that tracks HIV patients' clinic visits. The AdaPT-R case study demonstrates ODK 2's applicability to medical trials that are longitudinal and require patient follow-ups at specific locations. AdaPT-R was deployed in three clinics in Kisumu County, Kenya and two clinics in Migori County, Kenya. The five clinics combined serve approximately 65,000 patients. AdaPT-R is a randomized control trial studying 1,745 of the approximately 17,000 HIV patients. The study requires daily synchronization of a field worker's mobile device so that the medical *workflows* and study parameters can be updated appropriately. The non-linear *workflow* needed to ensure protocol adherence along with the inclusion of historical data to track lapses in care required the flexibility and functionality of ODK 2. The project also required an ODK 2 application to integrate with their existing medical records systems. The UCSF team uses the ODK 2 tool suite to collect and synchronize patient data daily to an *Aggregate* server. Once the data has been synchronized, it is combined with information from a separate medical records system to determine patient status. The patient information is then synchronized to the field worker's phone, where it can be used for data collection the next day.


Figure 3.5: Red Cross enumerator using ODK to distribute supplies in Belize after Hurricane Earl [20]. *(Photo courtesy of Ori Levari)*

### *3.5.4 Disaster Response*

The International Federation of Red Cross and Red Crescent (IFRC) is the world's largest humanitarian and development network; it has millions of volunteers in 190+ member National Societies. The diversity of use cases and business requirements of the National Societies demonstrates the need for a flexible, customizable system like ODK 2. A disaster response platform used by volunteers on the ground needs to be easily adaptable by a deployment architect since time is critical under these circumstances.

The IFRC created prototype disaster response software using ODK 1 for beneficiary registration and custom software for distributions. The IFRC then contacted us to figure out how ODK could be used to create a single mobile application to handle both registrations and distributions. ODK 1 was not suited to the task because of the missing features outlined in the beginning of the chapter. Bidirectional synchronization is required to maintain a consistent database of beneficiaries and delivered distributions across all devices and distribution

centers to prevent fraud, enable auditing, etc. User permissions are required to ensure that field workers see only the appropriate list of beneficiaries relevant to their work and cannot update data fields beyond the scope of their responsibilities. Their complex *workflow* that reuses certain data fields requires a more robust interface than the basic form-based navigation provided by ODK 1.

We worked with the IFRC to develop a prototype solution using a combination of *Survey* and *Tables* to build a custom data management application for disaster response. ODK 2's customizable *workflow*, user permissions, rule and adherence enforcement, disconnected operations, and eventual data synchronization are critical features needed to successfully coordinate distribution of cash and relief supplies. The IFRC's goal was to build a reference disaster response, disaster recovery, and crisis management application that utilizes ODK 2 frameworks to handle common data collection and field management tasks. This reference application could then be adapted to the diverse use cases of its 190 member societies.

### *3.5.5 Mosquito Infection Tracking*

The World Mosquito Program (WMP) [111] is an international research collaboration focused on reducing the spread of Dengue, Chikungunya, and Zika. The program operates across a variety of countries, languages, terrains, and worker skill levels. The WMP is developing a method to reduce the spread of these diseases by introducing the naturally occurring Wolbachia bacteria, which reduces the ability of mosquitoes to transmit viruses between people. Workers monitor mosquito reproduction at field sites until the bacteria infection rates reach a critical mass of the local mosquito population so that the propagation of the bacteria is self-sustaining. This monitoring requires repeated weekly visits to field collection sites. The ODK 2 frameworks are used to provide workers with an interactive map of site locations, historical data about each site, individualized instructions on what tasks to perform at which sites, and a data capture application. Their operational *workflow* requires ODK 2's bidirectional sync protocol to provide historical as well as timely data to their field workers during site visits to remote environments.

### *3.5.6 Tuberculosis Patient Records*

Mercy Corps' [91] effort to combat tuberculosis in Pakistan provided a case study for the ODK 2 frameworks' ability to serve rural medical facilities that maintain paper *workflows*. Mercy Corps maintains paper-based patient records in local clinics, which are subsequently aggregated and used to generate reports for administrators. Paper records are valuable in this environment since they are cheap, trusted, and easily understood. However, shipping paper records to central facilities and transcribing data from paper to digital systems hinders reporting efforts with significant time delays and financial burdens [2]. This project explored how to incorporate less expensive paper collection methods into a hybrid *workflow* to reduce data collection and aggregation times in disconnected environments.

# Chapter 4

# ODK 2 MOBILE FRAMEWORKS

Designing mobile data management systems for challenged network environments necessitates creating abstractions for *deployment architects* to configure and customize to their deployment contexts. ODK 2 frameworks are designed to be used in multiple application domains with varying deployment conditions to enable widespread use by global humanitarian organizations. ODK 2 frameworks need to handle varying data constraints and allow for customization of both the *workflow* and *dataflow* to meet deployment requirements. The frameworks' abstractions need to be flexible enough to handle various field conditions but simple enough for a non-developer to make adjustments in response to changing field conditions. We designed and built modular service-based software frameworks to empower organizations to create information management solutions that adjust to their constraints.

An essential aspect of ODK 2 design is that the frameworks be configurable in the field by *deployment architects* who have an understanding of field conditions, work processes, actions that *end-users* need to complete, and the data required for reporting purposes. There is often a usability problem when system designers attempt to use a single interface that targets both users with basic computer skills as well as users with advanced programming skills. A basic-skilled user can feel overwhelmed with complex interfaces, whereas a user with advanced programming skills can feel frustrated when the interfaces expose do not provide enough functionality. To overcome this problem, ODK 2 creates frameworks that have multiple abstractions that provide different methods of accessing similar functionality. Having multiple methods of achieving the same task through different *workflow* customization interfaces can assist users with different skill levels to configure ODK 2. Instead of a one-size-fits-all approach, ODK 2 creates multiple interfaces based on the amount of customization

that is needed (which can sometimes correlate to skill level). These multiple interfaces target different skill levels in an attempt to emulate the concept of design scaffolding [3, 113]) to enable users to learn more about the ODK 2 frameworks through the use of simple interfaces that are less confusing/intimidating. As an example, there are multiple different procedures to create a database table in ODK 2. One approach is for a *deployment architect* to design a questionnaire, and ODK 2 will automatically generate all required database tables to support gathering data with the questionnaire. This approach reduces the *deployment architect's* need to understand the complexity of database interfaces. However, if the *deployment architect* wants more control of the structure of the database, there is a second method to define a database table by simply supplying column names and column types in a spreadsheet (XLSX format) without having to write any SQL. The system takes this XLSX definition and automatically generates the necessary SQL statements. A third option for the *deployment architect* is to employ ODK 2's JavaScript functions to create the database tables programmatically with SQL statements.

As described in Section 3.4, the ODK 2 frameworks seek to satisfy an organization's information management needs for specific use case through the composition of narrowly focused tools rather than a single monolithic app. An app that tries to provide abstractions for every usage scenario can be overwhelming; therefore, ODK 2 software frameworks focus on providing specific functionality that can be used together with other functionalities from other ODK 2 tools. The various ODK 2 mobile frameworks are designed to smoothly transition between one another to give the *end-user* the feel of a unified application. As part of my research contributions, I designed and built multiple modular service-based application frameworks that comprise the ODK 2 tool suite. ODK 2 enhances the ability of organizations with limited technical capacity to build application-specific information services in disconnected environments. The frameworks are runtime-adaptable to various application domains through the use of configuration files and runtime programming languages with flexibility with regards to data input mechanisms, data models, *workflows*, *dataflows*, and visual appearances.

This chapter discusses the five ODK 2 Android frameworks that are part of my research contributions. Figure 3.4 shows the architecture of the six mobile device frameworks that when used together provide the full functionality of the ODK 2 tool suite. The five frameworks that are part of my dissertation are: *Tables*, the data curation framework; *Survey*, the question rendering and constraint verification framework; *Services*, the common data services framework; *Sensors*, the device connection framework; and *Submit*, the network communication framework. The sixth ODK 2 mobile framework is *Scan*, which was contributed by Dell et al. [36, 37]. *Scan's* creation is not part of my dissertation, but deployment experiences with *Scan* are discussed in the chapter on deployment experiences (Section 5.2.1).

As discussed in chapter 3, ODK 2 creates a runtime customizable user interface using HTML and JavaScript. ODK 2 frameworks are built as a combination of native Android user interfaces written in Java and web interfaces rendered using Android's WebKit. A 'WebKit' is a web browser that is embedded into an Android app's interface to enable an Android app to render web pages without the need to use a web browser (e.g., Chrome, Firefox). The usage of native Android user interfaces facilitates the access of Android system resources that cannot be accessed inside the WebKit because of system security. To enable organizations to be able to access some of the protected functionality when customizing their workflows, ODK 2 injects custom JavaScript objects that communicate with the compiled Android Java code to execute the functionality using Android system constructs. This JavaScript to Android Java abstraction is one of the significant abstractions that simplifies an organization's ability to customize ODK 2 by decreasing the amount of software code that needs to be written to customize workflows. Another key abstraction that simplifies customization is the centralization of database service to a single shared-interface in order to enforce consistent semantics and data-access restrictions. Section 4.3 describes the functionalities provided by *Services* to the other ODK 2 mobile frameworks that include the database service, a lightweight web server, user authentication, framework preferences, and the data synchronization protocol.

## *4.1 Tables*

The *Tables* framework [21, 68] is a general-purpose data management app for Android devices that enables organizations to create custom *workflows* for *end-users* to interact with sets of data. The *Tables* name comes from the fact that the underlying abstraction to store data in ODK 2 is tabular data stored in a database. *Tables* is designed to be a flexible information management solution for a variety of use cases where previously collected data is often revisited and updated, such as in logistics management and/or public health. The flexibility it provides for accessing and rendering data creates a framework that organizations can extend to implement customized information services for a variety of use cases. *Tables* provides workers in the field the ability to collect, manipulate, and view multiple rows of data on their Android devices. Views of the data can be either tabular or graphical, depending on the context and what is appropriate for an *end-user's* task.

When creating the *Tables* framework, we sought to establish a design space between spreadsheets and relational databases by borrowing the best elements of each and omitting more general features that would lead to an overly complex user interface. There are several smartphone apps that implement traditional spreadsheets (i.e., Excel-like functionality where formulas can be attached to individual cells). The name *Tables* was specifically chosen to highlight the distinction from spreadsheets and emphasize the similarity to relational database tables. Formulas are all column-based rather than cell-based, as in traditional spreadsheets. *Tables* aims to help organizations that need to have customized information applications in contexts that necessitate a workforce having limited technical knowledge. *Tables* presents a user interface optimized for the smaller screens of mobile devices and provides customization capabilities for *deployment architects* to easily configure the app for their use case.

*Tables* enables users to enter and curate tabular data on Android devices and allows for the expression of relations between data sets, thus enabling cross-indexed data. The flexibility provided by the *Tables* framework dramatically increases the usability of the ODK 2 tool

suite, as organizations are able to incorporate multiple pieces of data when encoding complex *workflows* and decision logic. Presenting the *end-user* with *workflows* that include previously collected data outside the directed question *workflow* of *Survey* (described in Section 4.2) expands ODK 2's adaptability multiple tasks *workflows* that a field worker might need to chose which to perform based on the data being presented. *Tables* provides several built-in views (e.g., table view, list view, map view, graph view) that can pull data from, and link to, other data tables, so that *deployment architects* can form an integrated mobile information management application, rather than a set of loosely connected (or completely disconnected) data tables. *End-users* can explore their tabular data through a variety of views enabled by a collection of HTML and JavaScript files. *Tables* use of JavaScript makes customization easy and flexible without requiring recompilation. *Tables* serves as a customizable *workflow* framework for viewing multiple pieces of data with the ability to link to other views or launch other ODK 2 frameworks. Instead of building a single monolithic app, ODK 2 uses a modular approach and multiple frameworks with different functionalities. *Tables* enables a *deployment architect* to create a *workflow* for *end-users* that includes viewing and curating data. The data curation *workflow* can integrate the launching of other ODK 2 frameworks to enable an *end-user* to leverage the different functionalities of the other frameworks.

### *4.1.1 Requirements*

The requirements for the *Tables* framework evolved from a set of use cases for ICTD projects deployed by organizations that were using ODK 1 tools [22, 21, 68]. Based on feedback from these users and their feature requests, as well as our own deployment experiences, we identified a common model for these use cases – namely, an organization with many field workers who need to share a single data set consisting of multiple tables. The workers may need to view and update this data set in the field, and managers need to be able to restrict access so that *end-users* only see and edit the parts that are relevant to them. Additionally, we conducted an online survey of ODK users where 85 respondents from around the world provided information on what were the most important features that would improve their

experience with ODK, as well as what missing functionality prevented their adoption of the current ODK 1 toolset for some of their projects. When asked to rate how important it was to "Keep data on a mobile device synchronized with a server" on a scale of 1-"Not Important" to 5-"Very Important," 55% of the respondents gave a rating of 5 with a mean of 4 and standard deviation of 1.24. Informed by the answers to questions such as these, we derived the following list of requirements [21]:

- ***Add, delete, search, and update existing data, as well as custom queries to extract useful information.*** *The ability to interact with the data, rather than merely collect it, was one of the most desired features requested by respondents to the online survey. Moreover, data tables can be linked (or joined, in database parlance) to connect two columns in two different tables. The ability to import large amounts of data to build a new table from a standard spreadsheet file (e.g., CSV) and export data to the same file formats for easy interchange with other tools was also cited. Finally, the inability to pre-load an editable data set, possibly using the structured ODK Collect form rendering tool, was the most common reason given for an organization choosing not to adopt ODK.*
- ***Share data across devices and keep the ensemble synchronized.*** *Besides simply backing up data, it should be possible for multiple devices to access and modify the same set of data (with subsets keyed to different clients), and the data should stay in sync as tightly as connectivity limitations permit. Conflict resolution will most likely require human judgment, and a good user interface is essential for accomplishing this task.*
- ***View data in multiple ways.*** *Including standard methods to view data (e.g., spreadsheet, list, map, and graph views) is important, but users always want more customization abilities to allow for a particular emphasis or a different subset of information. It is important that the views be able to be customized without having to recompile the app. On the back end, the ability to aggregate and summarize data collected from multiple field workers was rated as "Very Important" by 60% of respondents.*

- ***Robustness to sporadic network connectivity.*** *The application should never rely on connectivity being available and must be able to cache data from the server as well as new data or edits entered by the user until connectivity is established. The synchronization protocol must deal gracefully with limited or no connectivity.*
- ***Easy to deploy and maintain yet flexible enough to be customizable to different needs.*** *Many organizations in the developing world do not have the ability to take on sophisticated IT responsibilities. The basic solution should be simple for less technical users to set up and keep running. However, it must be possible to customize as a project and the workforce matures using standard web technologies. Training costs must be kept low and be incremental as more advanced features are exploited or developed.*
- ***Provide moderately fine-grained access control that allows both table and row-level permissions.*** *Administrators should be able to restrict access to tables and metadata as well as be able to easily create subsets of data so that each subset can be made accessible for a particular user or group of users. As most ODK users are already familiar with the data form model, their finest-grain size is a row of a table representing one instance of a form. The forms used for editing the data include finer-grain controls for restricting access to individual data fields. There must also be a simple mechanism for administrators to create groups of users and set access controls for each group.*
- ***Maintain a complete history of changes to a table on the server.*** *When rows are created, updated, or deleted, the history of these changes should be saved and made accessible to administrators for post-analysis of any problems that may arise.*
- ***Scalability in terms of data and number of users.*** *It must be possible to efficiently handle (UI interactions, file import/export, synchronization, etc.) moderately sized datasets (1000s of rows in tables with 10s of columns). Additionally, deployments involving hundreds of workers and client devices must not pose special challenges to the server infrastructure.*

- ***Extensibility to other server infrastructure.*** *To fulfill the modular philosophy of ODK it should be possible to adapt the server storage and synchronization protocol to different cloud and database services using a RESTful HTTP API. Similarly, new clients (e.g., an embedded monitoring sensor) should be able to implement a small set of interactions to a server using the same API.*

### *4.1.2 Framework*

*Tables's* user interface is optimized for the smaller screens of mobile devices and provides customization capabilities such as grouping, sorting, and updating the data. Many features are configured only once during setup by a *deployment architect*, who configures *Tables* for a particular application (or use case). *Tables* provides a number of built-in views and allows *deployment architects* to create their own views with HTML and JavaScript. To form an integrated data management application, *Tables'* views can pull data from, and link to, other data tables – thereby creating a "join" in the vocabulary of relational databases. However, *end-users* (e.g., field workers) do not write queries in SQL. Instead, queries can be initiated through the *Tables* search box, where terms are logically ANDed and applied only to the rows of that table.

*Tables* provides a number of different data views and allows users to choose which view they prefer on a per-table basis. A list view (see Figure 4.1a) is specified by HTML and JavaScript files that enables *deployment architects* to pick a subset of the row's data to be displayed for each list choice. *Tables* automatically provides a detailed view of a particular row's data fields that can be customized using HTML and JavaScript (see Figure 4.1b). *Tables* also has a graphical map (see Figure 4.1c) and fully customizable HTML pages. The map view uses Google Maps. The graph view uses the d3.js package [33] allowing *deployment architects* to generate custom graphs from the data. The detailed view also uses HTML and JavaScript files to enable a *deployment architect* to specify how to display the row's data fields. Detail view templates are provided to the *deployment architect* to automatically display all data in a row sorted by alphabetical order of the column name. To provide

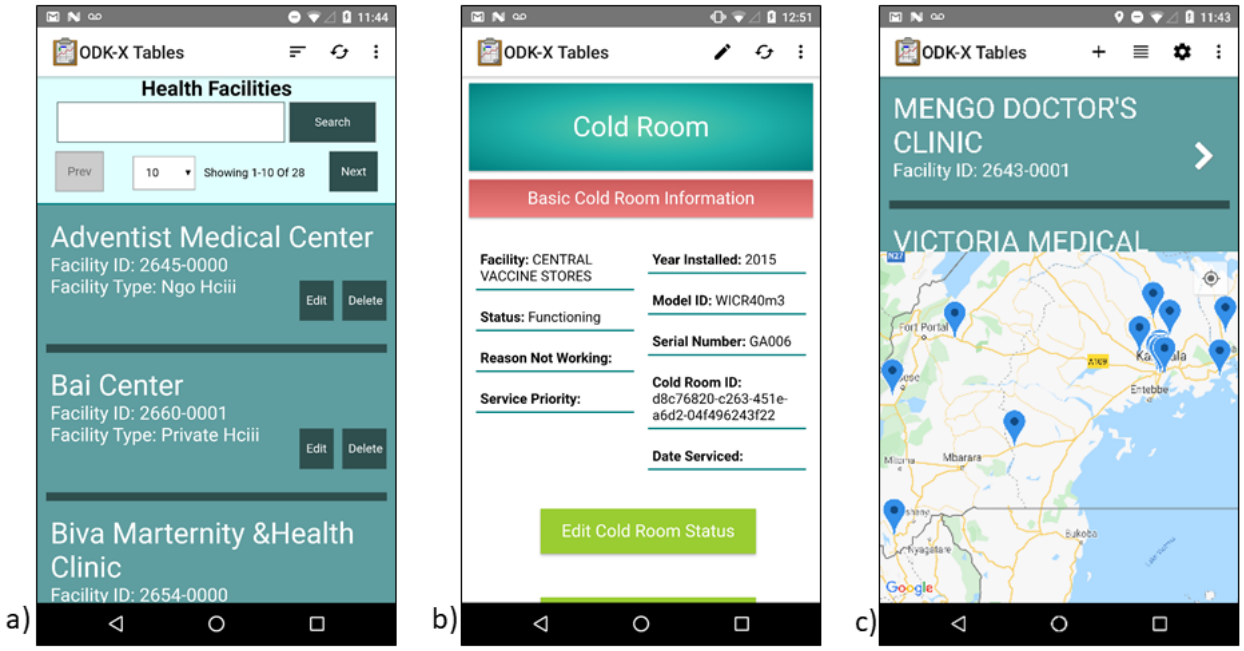



Figure 4.1: Figure a (left) is a list view of cold chain facilities. Figure b (center) is a detail view of a cold room facility. Figure c (right) is a map view of locations of cold chain facilities. The three *Tables* screen captures are from the vaccine cold chain inventory application (see Section 5.4)

flexibility in *workflow* design, *Tables* enables multiple HTML and JavaScript files to define multiple list and detail views for the same data table. Having an assortment of options for viewing data enables *end-users* to have access to the data in a variety of ways, thus allowing the adaptability of system views to the task being performed by the *end-user*.

To assist *deployment architects* in their customization efforts, *Tables* provides templates that require simple JavaScript commands for accessing data and basic HTML for formatting. To provide access to Tables' rendering framework, two objects ("data" and "control") pass information from the Android app's Java code to the custom JavaScript rendering the user interface in a WebView. With these two JavaScript objects, the custom view can access data and open new views to guide an *end-user* through a *workflow*. Additionally, the data object allows access to data by row number and column name. The control object can be used to

query for additional information (from any table), with the same user search format as used in the app's search box.

*Tables* has several options available to a field worker to add or modify data. *Tables* can automatically launch a *Survey* form for the *end-user* to enter data using an organization's pre-defined *workflow*. By utilizing *Survey* to enter and edit data, the *end-user* is constrained by a predefined *workflow* that supports correctness constraints as opposed to having users enter free-form text, which can be highly error-prone. The shared database presents users with a seamless data management application experience by smoothly transitioning between the tools (e.g., from *Tables* to *Survey* and back). Additionally, users can launch *Survey* with a form using current values in a row as starting values. Pre-populating values with previously collected data was a feature requested by multiple organizations to decrease the amount of data entry by *end-users*. Similarly, entire rows can be edited through a dynamically generated multi-screen form; however, this dynamically generated functionality does not include pre-defined *workflows* optimized with constraints and skip logic by a *deployment architect*.

We experimented with adding multiple types of functionalities to *Tables* to help users sort and curate their data. While useful for the "imagined" use-cases, in the end, organizations were less concerned with these exploration features and requested we keep *Tables* more straightforward for users. Organizations wanted to simplify training and keep the worker focused on their pre-defined *workflows*. One example of exploration was the concept of 'categories.' When viewing data in a large table, users often want to organize it into categories, such as grouping patient data by patient or grouping market prices by the type of goods. The 'categories' concept targeted tracking historical data. For example, if an organization is collecting temperature data every hour for several refrigerators, they would likely want to be able to see: a) the most recent data for all refrigerators and b) all data for a particular refrigerator. Relational databases provide this functionality with "GROUP BY" and "ORDER BY" statements in SQL. *Tables* allowed users to designate columns as "index columns," meaning they are used to group data or "collections." If they have marked at least one index column, a user could have two views of their data: an overview, which shows one item from

each group; and a collection view, which shows all the items in one particular group. A user was also able to designate a "sort column," which is used to determine which row from each group is shown in the overview – a combination of "ORDER BY" and "FIRST()" in SQL – (e.g., refrigerator data could be sorted by a timestamp column, so that the most recent reading for each refrigerator is shown in the overview). Collections might not be the only way a user wants to browse their data, so users were also able to search a table with column-value pairs (logically ANDed) and obtain a subset of the table that matches the search terms. In essence, we experimented with mapping some of the most common SQL constructs in simple column properties and search conditions.

### *4.1.3 Performance*

While the core functionality of *Tables* was useful to organizations, the data set size limited *Tables'* overall performance because of the time needed to retrieve and render a table. Originally, the table view was implemented using Android's 'TableLayout'; however, the user interface performed too slowly when rendering a large table of data. The slowness for large tables was caused by a Java object having to be constructed for each cell, causing the display time to grow with the size of the table. Therefore, we created our own custom table view that scales better with varying table sizes. Since we pass large data objects between Java and WebViews and then render them with JavaScript, this was another area of performance concern. Both of these aspects of rendering views were important because they affect the user interface and responsiveness of the app.

The time it takes to load a table is affected by the size of the table, which is defined by the product of the number of rows and columns. A two-column, one-row table takes approximately one second to load. A 700 row, 60 column table takes less than three seconds to load, and a 1400 row, 42 column table takes less than four seconds. Once loaded, the view is highly responsive to being scrolled and selections can be made immediately because the computation cost is proportional to what is displayed rather than to the underlying table size. Operations such as queries are still proportional to the size of the table but are not tied

to the display and rendering. Thus, querying the 1400 row table and restricting its contents to a single entity takes less than one second.

The two large tables in these tests were constructed from vaccine cold chain inventory data from an African country (see Section 5.4). As performance is highly variable between mobile devices, these heuristics are not intended as true benchmarks. They are provided only to demonstrate that relatively large tables are quite feasibly managed by the framework. Additionally, mobile devices have significantly improved their performance because of technological advances since the 2012 date of these tests. Furthermore, in practice, we rarely see larger tables as most of the time the data ends up being divided by regions for data management reasons.

Graph and map views are more forgiving in terms of required performance as they are a shift in view mode for which the user is prepared to wait an extra second or two. Since *Tables* uses external libraries and services for graph and map views, the performance is dependent on the speed of the external services. For 1000-2000 data rows, we observed performance on the order of 1-2 seconds for graphs generated by d3. For maps, the variability is much higher due to obtaining the map tiles over the Internet; however, if the tiles are present in the cache, performance is similar to that of Google Maps app since it using Google Maps infrastructure.

### *4.1.4 Summary*

The *Tables* framework is designed to be a flexible information management platform for a variety of use cases, including logistics management, public health, and environmental monitoring, where previously collected data is often revisited and updated. *Tables* helps fulfil the following ODK 2 design goals: 'the presentation layer must be independent of the navigation' and 'validation logic and the presentation layer must be customizable without recompiling the Android apps via HTML, JavaScript, and CSS.' It also provides features to meet the following key requirements of ODK 2: 'Complex / Non-Linear Workflows', 'Linking Longitudinal Data to Collected Data', 'Custom Apps Built with a Runtime Language', and 'Custom

Data Types that Update Multiple Fields in a Single User Action.' *Tables* focuses on providing functionality to display, interact, curate, and update a data set. *End-users* can explore the data through a variety of built-in views or create custom views using JavaScript/HTML, thus making customization easy and flexible without requiring recompilation. Reusable templates for viewing and organizing data through the framework's interface make it easier for organizations to design custom views to display and summarize data. *Tables'* performance results show it's well suited for scenarios with moderately sized datasets and deployments. *Tables* provides a customizable data curation framework that enables organizations to specify workflows in ODK 2 that can leverage the functionality of *Tables* and integrate seamlessly with other functionalities from other ODK 2 tools.

### *4.2 Survey*

*Survey* improves user experience through interactive, non-linear navigation allowing organizations to customize *workflow* complexity for differing levels of expertise. *Survey*'s ability to express nonlinear *workflow* logic and add additional functionality at runtime enables more flexibility for various application domains. ODK 2's *Survey* is a more flexible questionnaire-rendering tool that complements ODK 1's *Collect* tool. *Survey* improves ODK's malleability by solving some of the ODK 1's questionnaire-rendering design limitations. Like *Collect*, *Survey* is a question-rendering and constraint-verification framework that focuses on collecting and editing data through the use of questions. *End-users* progress through question-prompts as they capture data based on *Survey*'s form model. To easily incorporate new features and express complex *workflows*, ODK 2 tool suite abandoned the XForm standard.

*Survey* supports rich data types, including locations, barcodes, photos, audio recordings, etc. Data types (e.g., photos) that require large amounts of space to store are stored as a file. The database row saves the filename and path to enable the file to be loaded when it needs to be rendered. Using a common format, such as a file, that exists outside ODK 2's internal database, allows the ODK 2 framework apps to leverage other Android apps. For example, a new photo can be created by issuing an Android Intent to a camera app. The

camera app informs the ODK 2 framework of the file location that is an output of the user's interaction with the camera app.

*Survey* increases an organization's ability to customize an application by implementing the user interface with a runtime scripting language that allows for customizing without recompiling. *Survey* provides an assortment of default JavaScript/HTML question widgets that encapsulate the rendering, event handling, and *workflow* logic. *Survey* includes the suite of standard ODK 1 question prompts; however, these question widgets are instantiated at runtime to incorporate rendering and business logic customizations (e.g., visibility and value constraints). Organizations can extend these standard widgets at runtime to change the rendering or *workflow* logic itself (e.g., complex types, value constraints), to customize event handling, and to define new question widgets. For example, *deployment architects* can now define custom question types with multiple input values but with a single question prompt (e.g., blood pressure, pulse/oxygen), since question widgets are modifiable at runtime instead of being limited to the one-question/one-value model of ODK 1's JavaRosa XForm.

To simplify questionaire design, the XLSForm syntax from ODK 1 was restructured to reduce the number of spreadsheet columns and move complex programming syntax into a single column. The aim was to make it easier for *deployment architects* to initially learn the simpler questionnaire structure and then ramp up their knowledge as they start to need more advanced features. The entire flow of the *Survey* app, from opening screen to the outline/index view of a form, can be redefined or extended, thereby enabling more customization than the less flexible XForm syntax of ODK 1's *Collect*. JavaScript and HTML were chosen because they are common web languages with wide dissemination compared to XForm knowledge. Using common languages increases the number of qualified individuals that can be found to make domain-specific or application-specific customizations. Runtime customization permits a wider variety of customization possibilities than the XForm syntax of ODK 1, thus enabling *deployment architects* to learn the simpler *Survey* definition structure initially and then ramp up their scripting knowledge as they require more advanced features. Furthermore, *deployment architects* can use the new complex commands to express nonlinear

*workflow* logic using 'gotos' in the questionnaire definition. Buttons can be specified in *Survey* to perform user-directed form navigation expressed, effectively, as computed 'goto' (switch-like) statements. To help users keep complex *workflows* organized, the concept of 'sections' was added to enable logical groupings of questions. The concept of 'repeats' is replaced by the concept of 'subforms' to give users the full power of an independent questionnaire definition when dealing with repeated input sets.

*Survey* departs from the ODK 1 model by using runtime languages to define a suite of prompt widgets, rendering logic, event handling, and form navigation logic; this enables the entire flow of *Survey*, from the question prompt to the outline/index view of a form, to be redefined or extended at runtime. In *Collect*, changing the look-and-feel of a particular question type or extending the expression language (e.g., adding a count function) to express the user's business logic (e.g., visibility and value constraints) required changes to Java source code. This high barrier to change meant that we spent a significant amount of time refining our user interface because it needed to be generic enough to work for many use cases. It also created friction for the adoption of *Collect* because organizations lacked the skills or funding necessary to customize the tool. In contrast, *Survey* allows organizations to easily express their business logic and heavily customize the user interface for their specific use case through the use of JavaScript and HTML. The presentation layer can be easily customized by revising *Survey's* templates and CSS style sheets to create an organization-specific look-and-feel. *Deployment architects* can easily customize the user interface by specifying an alternative Handlebars template [60] in the form definition, causing the widget to render using the alternative template. *Survey's* JavaScript form interpreter, the use of open-source toolkits (e.g., Handlebars[60], Backbone[8]), and the greater worldwide prevalence of JavaScript and HTML coding skills make it easier for individuals and organizations to make domain-specific customizations.

For many organizations, the inability to easily customize complex *workflows* and presentation layers was an impairment that limited the usefulness of ODK 1. *Survey* helps to elevate the Open Data Kit Project from a project focused on data collection, to a project

that is capable of assisting users to make complicated decisions in the field. *Survey* improves user experience through interactive, non-linear navigation, allowing organizations to customize interface complexity for differing levels of expertise. *Survey* helps fulfil the following ODK 2 design goals: 1)'Workflow navigation should use intuitive procedural constructs, function independent of data validation, and allow for user-directed navigation of the form;' 2)'the presentation layer must be independent of the navigation and validation logic;' 3)'the presentation layer must be customizable without recompiling the Android apps via HTML, JavaScript, and CSS;' 4)'partial validation of collected data should be possible and the validation logic should be able to be dynamic;' 5)'local storage should be robust and performant for data curation and for longitudinal survey workflows using a relational data model;' and 6)'multiple data collection forms should be able to modify data within a single, shared, data table.' Survey is a runtime customizable question-rendering and constraint-verification framework that enables organizations to digitize interactive, non-linear *workflows*.

## *4.3 Services*

*Services* [20] provides the shared data storage and synchronization frameworks that are used by all ODK 2 apps running on the Android device. One of the key components of *Services* is the database service that abstracts database access to a single shared-interface to enforce consistent semantics and data-access restrictions. *Services* exposes interfaces to other shared functionalities, including a lightweight web server, user authentication, framework preferences, and the data synchronization protocol.

### *4.3.1 Services Architecture*

In the first iteration of ODK 2, the mobile frameworks were designed as completely independent Android apps that directly accessed a single shared Android database. An Android database is based on SQLite [131] and stores the data in a file. The shared database presented users with a consistent data management application experience as all the tools (e.g., *Tables*, *Survey*) had access to the same data stored in the database. Any change made in one

ODK 2 tool would immediately be visible in the other tool. Originally, each mobile framework included its own logic for accessing the database. From within the WebKit, the shared Android database was directly accessed via WebSQL. Subsequent stress tests of ODK 2 app transitions exposed an instability between the WebKit and the Java layer's Android user interface life cycle because of a race condition. The rate of deadlock occurrences varied by device model and OS version. Interleaving multiple Android user interface life cycle events during database transactions resulted in deadlock if the outgoing activity did not release the database transaction. This was complicated by ODK 2's use of WebKits for rendering the user interface. The WebKit performs synchronous calls to the Java layer, but only asynchronous calls from the Java layer to the WebKit are provided in Android. Android life cycle event handlers are synchronous and provide no mechanism for an asynchronous resumption of life cycle transitions.

To resolve this deadlock issue, the ODK 2 tools were re-architected to access the database through a single interface proved by *Services*. The *Services* app provides a centralized database service that facilitates 1) initialization that automatically pre-populates tables with data rows from configuration files and 2) a reduction in overhead when opening an already-initialized database since *Services* maintains knowledge of the database state. After creating the database service, we uncovered deficiencies within the Android database implementation, including reference counting issues, database locking, and the database connection object being stored in thread-local storage. These deficiencies led us to use a custom build of the SQLite database with a Java wrapper to 1) eliminate the need to address behavioral differences across different Android operating system versions, 2) enable direct management of connections, 3) speed up the opening of database connections, 4) enable the use of write-ahead-logging, and 5) allow changes to the native interface to minimize memory allocations.

While *Services* exposes Android content providers for a few basic static queries, the primary access to stored data is through the database service. The database service presents a restrictive programming interface that does not expose database transactions or cursors. Instead, it exposes atomic-update primitives and queries that return snapshot-in-time result

sets. This atomicity eliminates the possibility of a misbehaving client causing deadlock or leaking cursor resources. We chose the Android service component over a content provider because 1) insert/update/delete operations would not capture the atomic update primitives used to simplify database management, 2) the full capabilities of a SQL query cannot be exposed without introducing our own expression language embedded within that URL as opposed to a service API, and 3) the ability to apply row-level access control cannot be expressed through a single content provider. To use a content provider would require a custom-defined URL parser for SQL syntax. This would introduce custom non-standardized parsers causing a future barrier similar to the custom JavaRosa XForm definition and parser in ODK 1. One issue we encountered was that Android service components have a 1 MB size limit on remote-procedure calls, which is insufficient for large result sets. To address the size limit, we implemented a primitive transport-level chunking interface using a client-side proxy to re-assemble chunks and only expose a higher-level abstraction to the tools. The client-side proxy provided additional benefits, such as caching and typed exception throwing across processes.

To create a unified user experience, all application-level settings were consolidated into the *Services* app. Additionally, the local web service and the synchronization service were centralized into *Services*. By placing the synchronization service within the same trusted layer as the database service, we could maintain a chain of custody for the authenticated user identity gained through the interactions with the cloud component. This authenticated user identity could then be used to enforce row-level and table-level permissions inside the database service. More specifically, it enabled us to 1) verify the active user's identity, 2) fetch that user's permissions from the cloud component, and 3) cache the user's identity and permissions within this trusted services layer, where those permissions could be used within the database service to enforce the visibility and modification constraints on the data tables.

*Services* also simplifies tool updates. By effectively consolidating the application's persistent state behind this service layer, individual ODK 2 frameworks can be upgraded in a rolling-update fashion without affecting the persistent state. If the data representation

(schema) needs to be changed, only a single update of the *Services* tool is required. Other benefits include defining our own service application programming interface makes it easier to port ODK software to other platforms and increased code reuse allows for improved testing and stability.

*Experiments*

Performance is a persistent concern for organizations deploying in low-resource contexts, as cheaper Android devices often do not have the processing power of Android devices sold in high-resource contexts. To evaluate the performance of the revised service-oriented architecture, six different variants of ODK 2 'apps' were created to compare performance timings.

The six test 'apps' differed by type of database, by whether an Android service was used, and by whether the logic was separated into two different APKs or combined into one. We performed the tests on three different Android devices, across a spectrum of performance capabilities: a ˜$40 Vodaphone purchased in Kenya, a Nexus 7 tablet (a device commonly used by our field partners), and a Nexus 6 phone. We used Android OS versions commonly found in our target regions.

The setup used to evaluate performance consisted of an ODK 2 data management application and automated Android user interface tests. The application queried 200 rows of a 3000-row data set in succession until 100 iterations had been performed. Three timestamps were recorded by the application 1) before the query was issued, 2) when the query results returned, and 3) when the framework finished rendering the user interface elements on the screen. Table 4.1 presents the average query turnaround times, the average WebKit rendering times, and the overall (full round-trip) times. It shows queries performed through the service programming interface with the service and caller residing in separate APKs (the standard ODK 2 configuration), queries performed through the service API with the service and caller bundled into the same APK, and queries performed directly against the database without any service intermediary. These test results show that restructuring our tools to use a service-oriented architecture (SOA) did not incur prohibitive performance penalties.

Table 4.1: Database Query Timing Comparison of Different Implementations on three different Android Devices. Service-Oriented Architecture in Multiple APKs, vs Service-Oriented Architecture in Single APK, vs Direct Database Access. [20]

| ***Device & Operating System*** | ***Test Setup*** | ***Database Type*** | ***Query Avg.*** *(ms)* | *Query Std. Dev* | ***Webkit Render Avg.*** *(ms)* | ***Full RTT Avg.*** *(ms)* | *Full RTT Coeff. of Var.* |
|---|---|---|---|---|---|---|---|
| Vodafone v4.4.2 | SOA Multi APKs | custom | 2572 | 60 | 1474 | 4046 | 2.98 |
| Vodafone v4.4.2 | SOA Single APK | custom | 2635 | 222 | 1475 | 4110 | 6.81 |
| Vodafone v4.4.2 | Direct DB Access | custom | 2518 | 62 | 1446 | 3964 | 3.21 |
| Vodafone v4.4.2 | SOA Multi APKs | default | 2685 | 56 | 1509 | 4194 | 2.75 |
| Vodafone v4.4.2 | SOA Single APK | default | 2692 | 74 | 1539 | 4231 | 3.28 |
| Vodafone v4.4.2 | Direct DB Access | default | 2631 | 76 | 1504 | 4135 | 3.17 |
| Nexus 7 v4.4.4 | SOA Multi APKs | custom | 881 | 102 | 1049 | 1930 | 8.02 |
| Nexus 7 v4.4.4 | SOA Single APK | custom | 821 | 80 | 1063 | 1884 | 7.86 |
| Nexus 7 v4.4.4 | Direct DB Access | custom | 613 | 77 | 974 | 1587 | 8.58 |
| Nexus 7 v4.4.4 | SOA Multi APKs | default | 993 | 92 | 1047 | 2040 | 6.78 |
| Nexus 7 v4.4.4 | SOA Single APK | default | 1083 | 106 | 1067 | 2150 | 7.75 |
| Nexus 7 v4.4.4 | Direct DB Access | default | 1358 | 209 | 1043 | 2400 | 9.89 |
| Nexus 6 v5.1 | SOA Multi APKs | custom | 402 | 52 | 944 | 1347 | 17.63 |
| Nexus 6 v5.1 | SOA Single APK | custom | 425 | 53 | 968 | 1393 | 16.61 |
| Nexus 6 v5.1 | Direct DB Access | custom | 261 | 34 | 942 | 1202 | 17.87 |
| Nexus 6 v5.1 | SOA Multi APKs | default | 615 | 84 | 956 | 1571 | 14.54 |
| Nexus 6 v5.1 | SOA Single APK | default | 663 | 101 | 995 | 1658 | 13.36 |
| Nexus 6 v5.1 | Direct DB Access | default | 386 | 41 | 942 | 1328 | 14.52 |

Table 4.2: Average Database Query Round Trip Time (RTT) vs Number of Rows Queried [20]

| ***Rows*** | ***Nexus 6*** ***Avg. RTT*** *(ms)* | *Nexus 6* *Std. Dev (ms)* | ***Vodafone*** ***Avg. RTT*** *(ms)* | *Vodafone* *Std. Dev (ms)* |
|---|---|---|---|---|
| 2 | 207 | 48 | 483 | 46 |
| 20 | 233 | 56 | 549 | 46 |
| 200 | 402 | 61 | 975 | 66 |
| 2000 | 2156 | 87 | 6008 | 139 |

Comparing the *SOA Multi APKs* timings and the *Direct DB Access* timings showed a ˜150 ms overhead with SOA. The ˜150 ms is well below the 940 ms rendering time incurred by the WebKit and is below the limit of perceivable delay.

An alternative architecture combines all tools (e.g., *Tables*, *Services*) required for a particular usage scenario into a single large multi-dex APK. The app needed to be multi-dex APK because the number of method references exceeded 65,536, which was the limit of the Android build architecture at the time. Using a modular app architecture avoids the issues with creating a single monolithic app. The timings results in Table 4.1 (comparing *SOA Multi APKs* timings and *SOA Single APK*) shows that there was a performance penalty, perhaps due to the increased proportion of available memory consumed by application code within a single APK. Combining only *Tables* and *Services* incurred a 25-50 ms penalty on a high-end device. If this penalty were driven by application code size, then a service-oriented architecture would be more performant than a monolithic APK that combines the ODK 2 frameworks.

To determine how the service-oriented architecture performance scales, we performed tests with varying database query sizes. The test setup consisted of an ODK 2 data management application and an automated Android user interface test infrastructure to execute the tests. The application repeatedly queried a 3000-row data set with a fixed query size (one of 2, 20, 200, or 2000 rows) until 100 iterations had been performed via the automated Android user interface tests. In this case, no user interface elements were rendered on the screen after the query results were returned to ensure that only the database interaction time

was measured. The application recorded a timestamp before the query was issued and when the query results were returned to the framework. Table 4.2 shows the average round-trip times and standard deviations for the database queries running on a Nexus 6 running the 5.1 version of the Android OS and a Vodafone running the 4.4.2 version of the Android OS. These results show that query times scale as expected with a lower-end device taking comparably longer than a more powerful device.

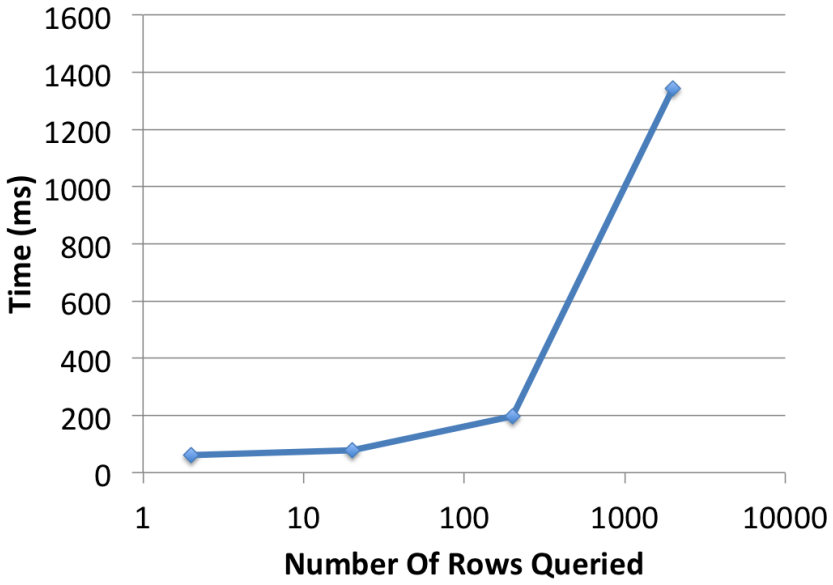



Figure 4.2: Nexus 6 Android version 5.1 Average Java To JavaScript Transfer Time vs. Number Of Rows Queried [20].

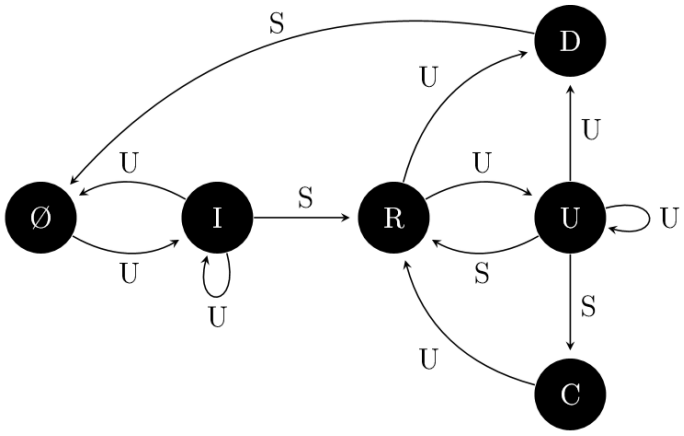
Finally, the amount of time spent marshaling database query results between the Java layer and the JavaScript layer was measured. To determine the transfer time between these layers, an ODK 2 application and an automated suite of Android user interface tests were used with a 3000-row data set. A timestamp was recorded in a log file when the Java layer obtained the query result and when the JavaScript layer received the query result. Figure 4.2 shows the average time difference between when the database query result was available in the Java layer and when the database query result was returned to the JavaScript layer for 100 test iterations. This data shows that the transfer time from the Java to JavaScript layer did not have a major performance impact.

### *4.3.2 Synchronization Protocol*

The synchronization protocol provides a mechanism to keep data on multiple devices synchronized to a master copy stored in the cloud. The underlying architecture consists of an HTTP API based on a Representational State Transfer (REST) architecture that defines a set of resources that can be manipulated using common HTTP verbs such as GET, PUT, and DELETE. The top-level resource of the HTTP API is a data table. Under the table are the sub-resources rows, columns, properties, and access control lists.

All operations are idempotent, meaning that the same action can be applied repeatedly and the value will be the same regardless of whether the action is applied once or multiple times. Idempotent operations simplify the synchronization protocol because the mobile clients do not have to be concerned about losing results or causing a bad state. Since Internet connectivity issues can be common in low-resource contexts, an idempotent design allows the action to be safely sent repeatedly to the server if a connectivity error occurs. Concurrent access to the rows of a table is handled with a custom, optimistic concurrency control locking system. Once a set of data changes is accepted, then all other changes from a different client will be rejected until that client synchronizes to the currently accepted changes. When a client requests to update a row in a table on the server, the server locks the entire table and executes the update, which includes updating the version of the row, called the "entity tag." Any concurrent updates must wait until the client requesting the change obtains the lock. When the second client obtains the lock, the second client request will attempt to alter the same row, but an error will be thrown because the second update's row version (i.e. "entity tag" or "etag") no longer matches the version of the master row on the server. The second update is consequently rejected, forcing the second client to resolve the conflict between the row pending changes on the client and the updated master row on the server before any further changes to the master row is allowed.

A challenge in building the synchronization protocol was designing the mobile client to track the synchronization state for every row of a table. To simplify the possible states in




**Key**

| States: | | Transitions: | |
|---|---|---|---|
| Ø | Null state | U | Transition made as a result of a user action. |
| I | Inserting | S | Transition made during synchronization. |
| R | Rest | | |
| U | Updating | | |
| D | Deleting | | |
| C | Conflicting | | |

Figure 4.3: ODK 2's basic synchronization state machine for a row to be synchronized with the server.

which data could exist, we created a finite state machine for the life cycle of a row. The state machine is illustrated in Figure 4.3; it has six possible states for a row [21]:

- ***Null:*** *the row does not exist*
- ***Inserting:*** *the row needs to be inserted on the server*
- ***Rest:*** *no changes to the row need to be communicated to the server*
- ***Updating:*** *the row needs to be updated on the server*
- ***Deleting:*** *the row needs to be deleted from the server*
- ***Conflicting:*** *both the local client and the server's copy of the row have changed and the user needs to resolve the conflict between them*

Each transition in the state machine is either the result of a user action or an action from the synchronization process. For example, when a user creates a new row on the phone, the row is moved from the 'Null' state, meaning it doesn't exist, to the 'Inserting' state. From the 'Inserting' state, there are three different possibilities: the user deletes the row, the user updates the row, or the synchronization process runs. The row stays in the same state if it is updated locally (including possible deletion). When the client synchronizes data to the server, the row will be inserted into the table on the server, and the row on the client will transition to the 'Rest' state to indicate all of the row's changes have been propagated to the server.

Figure 4.4 demonstrates an example sequence of interactions between *Services* and *Sync-Endpoint*. In this example, the server has three rows with changes that the mobile client does not know about. In addition, the mobile device has two changes that the server does not know about that include a row to insert and a row to update. First, the mobile device retrieves the latest entity tags that represent the current version number for both data and properties of the table. Entity tags are universally unique identifiers to prevent multiple mobile clients from having any version identifier collide. A collision is defined as two randomly generated versions strings having the same value, as a collision of version values would cause ODK to incorrectly assume the row values were the same since the version values were the same, even though the row values are different. For the purposes of this example, the versions are simply integers that are incremented every time a change is made to the table data or properties. However, in the synchronization protocol versions are universally unique identifiers to prevent version collision by different mobile clients making updates. By retrieving the latest version information, the mobile device can determine whether a change in the data or metadata has occurred since the last synchronization, as the version information would be different if there were changes. For this scenario, the mobile device starts at a data version of 2 and a properties version of 2. Since the server is at a data version of 5, the mobile device requests a list of differences that represents all changes to the rows of the table since the table was at data version 2. The server responds with a list of the three rows that have changed. The

mobile device then determines if the row is a new row or an update to an existing row and processes the action accordingly. With the mobile device data being up-to-date with the server, it is able to push its changes to the server. The server responds with the latest state of the row, including a new row version, which *Services* saves with its local row copy.

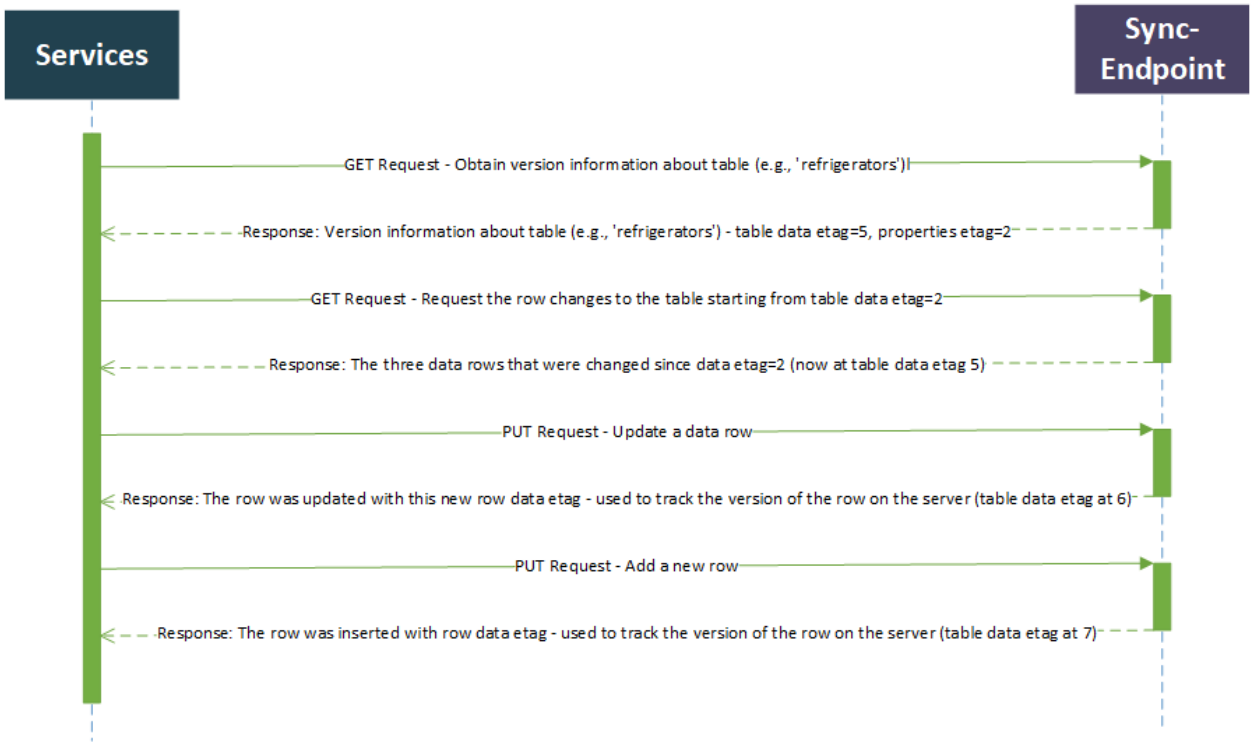



Figure 4.4: Example of an ODK 2's synchronization REST calls between a *Services* on a mobile device and *Sync-Endpoint*. The figure shows a sequence of interactions for the case with the following three changes that need to be synchronized: 1) the server has three rows with changes that the mobile device does not know about, 2) the mobile device has a row to insert, and 3) the mobile device has a row to update.

*Conflict Resolution*

In ODK 1, the data collected was not expected to change beyond corrections by the same person collecting the data. Conversely, a requirement for ODK 2 is to provide the ability for multiple users to update and change data over time. Updating data is a requirement to support use cases such as inventories and longitudinal studies. Since data is no longer immutable, ODK 2 relies upon the user to resolve conflicts that occur whenever two users concurrently update the same row in a table. There are several options for conflict resolution strategies that include options such as automatic versus manual resolution and server-side versus client-side resolution. Manual, client-side conflict resolution was chosen because of the following reasons:

- Established recent-modification conflict resolution techniques are inappropriate or difficult to apply across devices that may not be time-synchronized and are disconnected from the network for extended periods of time. Limited guarantees about the device's time accuracy and data delivery deadlines will require a complex protocol to account for multiple devices having inaccurate system times while operating disconnected performing conflicting data updates. Multiple *end-users* creating conflicting data during disconnected operation make it challenging for an automated system to determine which value is correct. Furthermore, this resolution scheme needs to be designed without prior knowledge of the type of data being updated or an organization's workflow.
- Since ODK targets a diverse set of use cases and application domains, any assumptions built into an automatic resolution mechanism will likely be inappropriate for some domains.
- Accurately expressing domain-specific procedural rules to be applied during automatic conflict resolution is likely difficult for non-programmers. In an automatic resolution scheme, an administrator would need to configure rules, such as to always take the changes of a certain user or group over another, or to try and intelligently merge

the changes on a column-by-column basis. Creating a procedural rules engine would make the server complex to implement and would still likely lack the configuration possibilities that organizations might desire for their application.

- Client-side resolution is preferable because it involves the *end-user* reconciling conflicts. This is advantageous since the person working in the field is more likely to understand the semantics of the conflict and can better resolve the conflict when detected. This is preferable to leaving a conflict for a remote administrator to resolve at a later date who does not have the context or may not be able to determine the correct answer without actually physically being in the location.

To minimize the number of conflicts, updates are row-based to keep changes to small bundles. Conflicts are detected and resolved at the individual row level (in keeping with our row-based information model) between a row on the user's mobile device and a row on the server. Detecting conflicts at the row-level maximizes the system's ability to disseminate new and uncorrelated change across devices. Taking the case study of cold chain inventory updates (see Section 5.4) as an example, if updates are at a coarse granularity, such as the whole table or file, a conflict might be detected for two workers updating the status of vaccine refrigerators at different sites that do not overlap. By keeping conflict detection at the row level, multiple users can make updates to shared data. When multiple users update different data rows, the system detects that the changes are conflict-free as long as the same piece of shared data does not change. Cell-based conflicts would be an even smaller data unit that would reduce conflict detection further. However, in a single row, many cell values are often inter-related. We felt that too much context would be lost with a cell-level conflict approach, potentially causing errors in reconciliation due to lack of context with interrelated cell values in a row. By always keeping the server in a consistent state, there can only be conflicts between a row on the user's phone and a row on the server. Furthermore, the user who caused the conflict is likely to know how to resolve it, so putting the responsibility in that user's hands is a sensible approach.

### *4.3.3 Data Permissions*

*Services* establishes a user's identity via a successful login to a server that implements the ODK 2 Cloud-Endpoint standards. Thereafter, the user's identity and permissions are cached until the user clears his or her credentials or until the device is next synchronized with the cloud component. During this disconnected period of operation, successfully unlocking the device is considered sufficient to re-confirm a user's identity. As discussed in Section 3.4, there are four classes of users: (1) anonymous and/or unauthenticated users, (2) authenticated unprivileged users with permissions to read and modify a subset of the data, (3) authenticated superusers with permissions to read and modify all data, (4) authenticated administrators can read and modify all data, update user permissions, and change the configuration files that specify the data management application. Administrators and superusers can manage which authenticated users can see or modify which rows. Unprivileged users may be given access and modification rights to individual rows that unauthenticated or anonymous users cannot access and/or modify.

Adding the ability for a *deployment architect* to provide different functionality to different levels of users was a feature request from organizations with more complex *workflows*. With ODK 1, there were basically only data collectors, but when introducing data management applications in ODK 2, there was a recognition that not all *end-users* would have the same skill-level or job roles. Data permission filters were added to ODK 2 to facilitate different roles, job tasks, or different *workflows* needed to customize to an organization's mobile data application. To meet the requirements of various organization's workflows, ODK 2 has both 'Table-level' permissions and 'Row-level' permission to facilitate diverse use cases.

**Table-level Permissions -** Data permission filtering introduces the notion of a locked table. Only superusers and administrators can create and delete rows in locked tables. Anonymous, unverified, or ordinary users are unable to do so. A table property is used to specify that a table is locked. Two other table properties control the creation of a row. The first property specifies whether an anonymous or unverified user can create a row in the

table (this only applies if a table is not locked; it has no effect if the table is locked since row creation is prohibited for all but superusers and administrators). The second property specifies the type of row-level access filter to assign to this newly-created row. The three table properties are specified in the properties sheet of the XLSX file definition for the table and/or form workflow.

**Row-level Access Permission Filters -** Management of what users can view, modify, or delete an individual row is controlled by the five access filter columns and the row's sync status. Initially, a row that has not yet been synchronized to the server starts with full read/write/delete capabilities. User-access permissions are enforced on the row once the row is synchronized to the server. ODK 2's row-level permission design was based on Linux style permissions with the following four permission possibilities that can be assigned:

- **'r'** – User has read access to the row.
- **'w'** – User has the ability to modify data columns in the row (Deletion is not allowed).
- **'d'** – User has the ability to delete the row.
- **'p'** – User has the ability to modify the row-level access filter columns (metadata columns).

ODK 2 grants a user a combination of the four permission filters with the following possible combination of permission values:

- **Null** - Not visible and no access.
- **'r'** – Read-only access to the row.
- **'rw'** – Read and modify access to the row.
- **'rwd'** – Read, modify, and delete access to the row.
- '**rwdp'** – Read, modify, and delete access to the row, plus the ability to modify the row-level access filter columns.

Once the row is synchronized to the server, the five row-level access filter metadata columns listed below are used by *Services* to determine what user access permissions should be granted. The _ROW_OWNER metadata column is simply the name of the user who

is considered to be the row's owner. The group metadata columns _GROUP_MODIFY, _GROUP_PRIVILEGED, and _GROUP_READ_ONLY contain the name of the group with the access privileges. The five row-level access filter metadata columns are:

- **_ROW_OWNER** – this user has FULL privileges on this row.
- **_GROUP_READ_ONLY** – a user who is a member of this group will be able to read this row of data
- **_GROUP_MODIFY** – a user who is a member of this group will be able to read and modify this row of data but not delete it.
- **_GROUP_PRIVILEGED** – a user who is a member of this group will be able to read, modify, delete, and change privileges on this row of data.
- **_DEFAULT_ACCESS** – This column specifies the access to the row that is granted to all unprivileged users and other users whose privileges did not give access via another permission construct. The possible values for _DEFAULT_ACCESS are: 'HIDDEN', 'READ_ONLY,' 'MODIFY,' or 'FULL.'

To determine a user's access to the row *Services* examines whether the user name matches the _ROW_OWNER or whether or not a user is a member of a group listed in the _GROUP_PRIVILEGED, _GROUP_MODIFY, and _GROUP_READ_ONLY metadata column. If the user does not gain access via the four user and group metadata columns, then the row-level access for the user is governed by the value in the _DEFAULT_ACCESS metadata column and whether or not the table is locked, as follows:

Table 4.3: Row-level data access permission filter for locked and unlocked tables

| *_DEFAULT_ACCESS* | *unlocked table* | *locked table* |
|---|---|---|
| FULL | rwd | r |
| MODIFY | rw | r |
| READ_ONLY | r | r |
| HIDDEN | not visible | not visible |

Superusers and administrators always have full read/write/delete/permissions ('rwdp') capabilities on all rows, regardless of their row-level access filters and independent of the table's locked status.

### *4.3.4 Summary*

*Services* provides functionality common to all the ODK 2 mobile apps, thus helping to maintain a modular architecture and simplify disconnected operation. Smooth operation in disconnected environments is a core tenet of ODK 2's design. Its synchronization protocols and structures are designed to be resilient in extreme mobile networking conditions such as low bandwidth and high latency environments. ODK 2 replicates data to the mobile devices to enable the frameworks to preserve full functionality in disconnected environments while maintaining the feel of connectivity via data synchronization and user permission enforcement. *Services* contributes some of the key functionalities needed to meet ODK 2 design goals including 1)'Local storage should be robust and performant for data curation and longitudinal survey workflows using a relational data model;' 2)'User and group permissions are needed to limit data access;' 3)'Disconnected operation should be assumed as data must be able to be collected, queried, and stored without a reliable Internet connection –when the Internet becomes available, the framework and cloud components should efficiently replicate data across all devices;' and 4)'Cloud components should be able to fully configure the data management application remotely as well preserve a change log of all collected data.' For example, the database service provides a single interface that enables disconnected operation, enforcement of consistency semantics, and data-access restrictions. The synchronization service provides the functionality to both configure/update data management workflows remotely and replicate the data between the mobile devices and server.

### *4.4 Sensors*

One of ODK's strengths is enabling organizations to create 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 seek to expand mobile information management applications to include an even richer set of data types. The platform shift from traditional PCs to mobile devices with cloud services has created a need and opportunity to integrate mobile devices with external sensors and deploy applications in new settings. The ICTD research community has been investigating leveraging mobile phones as a sensing platform for in-situ and remote monitoring [12, 26, 27]. Despite the proliferation of mobile apps and devices, there are only a limited number of applications that make use of external sensing devices. Resource constraints often prevent the adoption of sensing applications in under-resourced environments because of the lack of skills to implement the software needed to communicate between mobile devices and external sensors. Because of these complications, including sensors in mobile data collection often poses several technical barriers that, if reduced, would enable more mobile information applications to leverage sensors for data collection across varied domains. *Sensors* is a device-connection framework that seeks to simplify the creation of sensing applications by creating a framework with flexibility for integrating external hardware sensors into an organization's data collection *workflow*. The *Sensors* framework contributes to two main areas of research: (1) reducing programming barriers [34, 86] and (2) making mobile sensing applications efficient [87, 130, 154].

Even though capturing sensor data eliminates many of the errors that plague traditional data collection techniques, such as manual form-filling, it is still not widely used in low-resource regions because of the 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, creating a user interface, and designing application control logic. Unfortunately, this level of programming expertise is usually

not readily available in low-resource regions or even in high-resource regions on projects undertaken by resource-limited organizations. The *Sensors* framework [19] aims to lower implementation barriers by simplifying the deployment of sensing applications by:

- *Creating a modular framework for adding new sensors by abstracting the management of discovery, communication channels, and data buffers. Integrating a new sensor should require adding only its data handling and configuration primitives.*
- *Providing a high-degree of isolation between applications and sensor-specific code. Applications should continue to function even if sensor-specific code contains errors or a sensor becomes inoperative.*
- *Understanding the tradeoffs of several architectural approaches, especially modularity and performance.*
- *Facilitating 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 operating system configuration.*

*Sensors* provides a modular framework for organizations to integrate third-party sensors into ODK 2 by simplifying both application and device-driver development with abstractions that separate responsibilities between the 'user application,' '*Sensors* framework,' and 'device-driver.' The *Sensors* framework provides a single sensing interface to 'user applications' for both built-in and external sensors to simplify and hide a large number of details involved in integrating sensors into a 'user application.' By providing a common interface that abstracts communication channels, the *Sensors* framework simplifies sensor integration by providing a formatted data stream regardless of how the sensor was connected.

The framework also provides a simple abstraction on which to develop and deploy user-level device-drivers on Android. A primary goal of the *Sensors* framework is to simplify adding sensors by minimizing the responsibilities of the device-driver, thereby reducing the amount of work needed to create a device-driver. While a device-driver abstraction is a

standard concept, the *Sensors'* framework includes features that make the development of device-drivers easier by handling sensor state (e.g., connection, buffered data, threading). By providing a common interface that abstracts communication channels, the *Sensors* framework simplifies sensor integration by providing a uniform data stream regardless of how the sensor is connected to an Android device. This stream of raw sensor data is then processed by a simple 'user-level Android apps' that acts as a device-driver that parses the raw data stream with specific sensor logic to produce formatted sensor data. Since the *Sensors* framework only requires driver developers to implement sensor-specific commands and data processing, it means that the device-drivers can be implemented as stateless processors of data. Keeping device-drivers stateless allows the driver programmer to keep code simple by avoiding managing multiple sensors within the driver code or having to run multiple apps for each sensor type. Additionally, by using *Sensors'* common abstractions, the user interface to interact with a sensor can be developed independently of the device-driver, thereby enabling software programmers to develop sensor device-drivers that are reusable in multiple different application scenarios. The details of the interfaces presented to 'user application' developers and 'device-driver' developers are discussed in Section 4.4.2.

### *4.4.1 Requirements*

The *Sensors* framework expands ODK by creating a framework to easily augment consumer Android devices with external sensing capabilities. *Sensors* is designed to support a variety of external sensors that vary by the type of data collected, the communication channel used to report readings, and the rate at which the sensor generates data. The *Sensors* framework 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 because more programming is required to interface and process the data stream as compared to built-in sensors. The *Sensors* framework focuses on ease-of-use with a particular focus on appropriateness to low-resource contexts.

The *Sensors* framework is designed to isolate 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. The modular framework design is possible because Android has extensive support for background processes and includes several built-in constructs for inter-application communication (IPC) between Android apps. Additionally, decomposing the system into modules enables more effective testing and code reuse, thereby improving overall system robustness. Robustness is particularly important for low-resource deployment settings since once a system is deployed in remote areas it becomes logistically difficult to update the device because of the costs, time, and complexity to reach all of the sensors distributed in remote locations. For the *Sensors* framework to successfully enable a sensor ecosystem it must be [19]:

- *easy to create sensor drivers, that is, minimizing the knowledge and amount of code required to create a driver,*
- *easy to integrate/reuse external sensors in a wide variety of applications,*
- *easy to deploy the framework and device drivers, shielding an end-user from the technical details of the sensing infrastructure,*
- *easy to upgrade the framework and sensor drivers,*
- *hard for bad driver code to damage the framework since Sensor Driver Developers may not be expert Android developers,*
- *easy for an Application User to discover available sensors through a streamlined user interface, and*
- *easy to manage communication channel details such as proper handling of dropped connections.*

### *4.4.2 Framework*

There are multiple 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 the 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.

The *Sensors* 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 the least technically proficient of the three and is only expected to be able to use applications on an Android device, similar to the *end-users* role described Section 1.1: Multi-Perspective Design. 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. Both the 'Application Developer' and the 'Sensor Driver Developer' operate in the 'programmer' role described in Section 1.1. 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 *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, unified interface reduces complexity since all external sensors as well as Android's built-in sensors are exposed through the common interface regardless of the communication medium used. The interface encapsulates communication and separates 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 *Sensors* consists of three apps: the User Application App, the *Sensors* Framework App, and the Sensor Driver Apps. The *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.

*Sensors* provides a common interface to access both built-in and external sensors connected over a variety of communication channels. Thus far, we have implemented channel managers for Bluetooth and USB. The USB Manager currently supports three USB protocols: Android's Accessory Development Kit (ADK) 2011, ADK 2012, and a USB Host serial channel. The *Sensors* framework interfaces with Arduino boards [6] over USB 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. *Sensors* also provides a convenient built-in sensor discovery mechanism that allows users to discover sensors and associate the appropriate driver with a sensor. Users who want to integrate external sensors with their mobile devices may download and install the *Sensors* Framework App and required Sensor Driver App from an app store such as Google Play. This facilitates the easy delivery of the application and driver updates to devices. *Sensors* provides abstractions that delin-

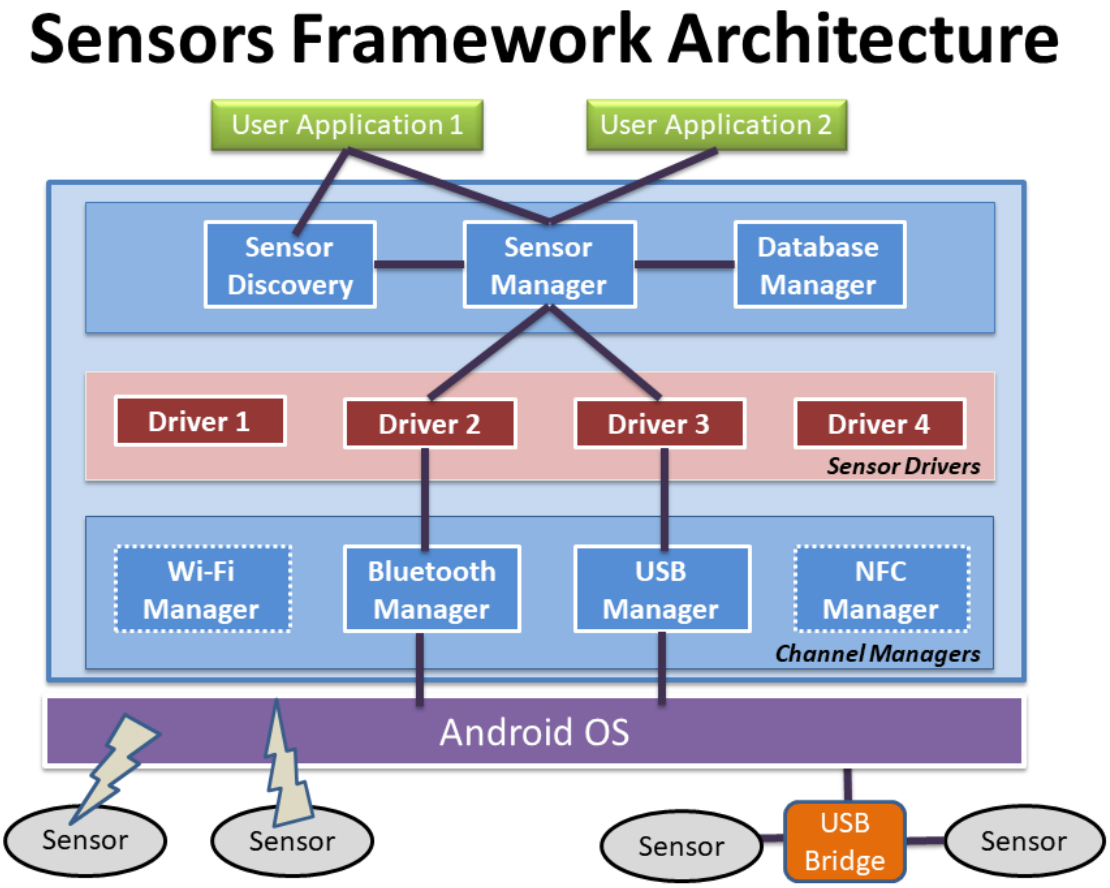



Figure 4.5: Architecture overview of the *Sensors* framework. The 'user applications' communicate to the Sensor Manager through the *Sensors* service interface. The Sensor Manger maintains the state and references to all sensors and the corresponding sensor drivers. The Channel Managers (e.g., USB Manager, Bluetooth Manager) abstract the communication channels and manage the state of connections on the communication channel to the Sensors.

eate application code from code that implements drivers for sensor-specific data processing. The sensor driver abstraction allows device drivers to be implemented in the user application space so that locked devices can be customized by *end-users*. The framework handles the data buffers and connection state for each sensor, which simplifies the device drivers. Separating application code from device driver code also allows the code bases to evolve independently.

The *Sensors* framework (shown in Figure 4.5) presents a common interface to all top-level user applications via the Service Interface. User Application Apps only need to implement the application-specific logic that handles processed sensor data received from the framework. 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. *Sensors* automatically manages this entire process for the 'Application Developer.' The current *Sensors* framework implementation supports communications over Bluetooth and USB, but *Sensors'* modular design and well- defined interfaces make it simple to add additional channel managers to support other communication methods such as NFC and Wi-Fi. To create a single unified sensing interface, the *Sensors* framework also exposes built-in Android sensors through its interface to create a single integration point for all sensors.

Sensor device 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 require 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 Developer' and the 'Sensor Driver Developer'.

*Sensors* framework is possible because Android has extensive support for background processes and includes several built-in constructs for inter-process communication (IPC) between Android apps. For instance, an Android Broadcast Receiver uses non-blocking message passing to communicate between applications; whereas, an Android Service communicates through a blocking inter-process communication mechanism. These constructs enable a comparison between a single-threaded asynchronous model (Android Broadcast Receiver) and a multi-threaded synchronous model (Android Service) for inter-application communication. The *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 automatically generates inter-process communication code to decompose objects into primitives that the operating system can marshal data as parcels across process boundaries.

*Sensor Framework Interface*

The *Sensors* framework interface abstracts many sensor-specific details and communication channel specifics. The *Sensors* Android service interface creates a common interface for user applications to leverage both built-in sensors and external sensors connected over multiple communication channels. The *Sensors* service interface functions available to Android apps to leverage are:

```
boolean sensorConnect(String id, boolean storeInDatabase);
void configure(String id, Bundle config);
boolean startSensor(String id);
boolean stopSensor(String id);
List<Bundle> getSensorData(String id, long maxNumReadings);
```

```
boolean isConnected(String id);
boolean isBusy(String id);
boolean hasSensor(String id);
```

To use the framework's interface it requires that the sensor's identification (SID) be used to identify which specific sensor in the framework to control. If applications do not know the SID, then the framework provides an interactive discovery process for users, freeing the application from implementing its own sensor discovery interface. To launch the *Sensors* interactive sensor discovery user interface, the application simply sends an Intent to the framework as described in Section 4.4.4. Once the application has the SID, it can connect to the sensor and begin retrieving sensor data by periodically calling the 'getSensorData' function of the framework interface. This function returns sensor readings to the application in a mapping of key-value pairs created by the Sensor Driver App processing raw sensor data. As an example, a temperature sensor's driver parses the sensor's data according to the messaging protocol of the sensor to extract multiple (or single) temperature readings. The sensor driver then formats the values and passes a list of Android bundles that contain the key-value pairs (i.e., key = "temperature", value = "¡value in Celsius¿") to the *Sensors* framework. The framework expects the User Application App to retrieve data in a timely fashion to clear the memory that is buffering the data. In cases where the User Application App does not plan to read the data immediately, the framework provides a second mechanism for retrieving sensor data stored in ODK 2's database.

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. *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.

*Sensor Driver Interface*

Sensor Drivers are designed to abstract sensor-specific control code from more general sensor management code. 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 sends all the raw data received from the sensor to the appropriate sensor driver to be processed. The sensor driver parses this sensor-specific data, transforming it into key-value pairs that can be easily processed by user applications that are outside the *Senors* framework. 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 sensor driver then encodes these key-value pairs according to the sensor's messaging protocol. This sensor encoded configuration data is then sent to the sensor via the Channel Manager. By having channel managers handle communication details, the device driver developer no longer needs to be aware of channel-specific protocols. Instead, the programmer simply implements the functionality required by the Sensor Driver interface. To create a Sensor Driver a programmer must implement the following interface:

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

Sensors 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. Functions 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.

In addition to allowing for easy reuse, the Sensor Driver design shields applications from changes in the communication protocol, configuration, or data type. *Sensors* framework modular design shields applications from driver or channel changes leading to more robust systems that are easier to maintain. The modular sensor driver abstraction 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).

### *4.4.3 Experiments*

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 and understand which Android inter-process communication (IPC) constructs to use to create our application-level (or user-level) sensor driver framework, we implemented three different

versions of the framework. As depicted in Figure 4.6, 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 Android Service 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 *Sensors* framework as needed. User applications remained the same across the three implementations, as their interface with the framework did not change. The Sensor Manager and Channel Managers communicate with sensor drivers either with function 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.

**V1: Drivers within the Framework** – The V1 version of the framework implements a design where all the sensor drivers software is included inside the *Sensors* framework app thus creating a single monolithic Android app. At runtime, the framework accesses individual sensor drivers from an in-memory map. In this architecture, since the drivers execute within the framework it should produce the best performance and provide a baseline against which to compare the other versions because the Android operating system does not have to pass data between processes. 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. Our goal is to build a system that uses established Android distribution and communication mechanisms to dynamically deliver and add drivers to the *Sensors* framework.

**V2: Driver Communication via Services** – The V2 version of the framework creates separate Android apps for each sensor driver and uses the Android Service construct for communication. 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 inter-processes. Android provides the AIDL to automatically generate the inter-process communication code based on the programming interface between the client and

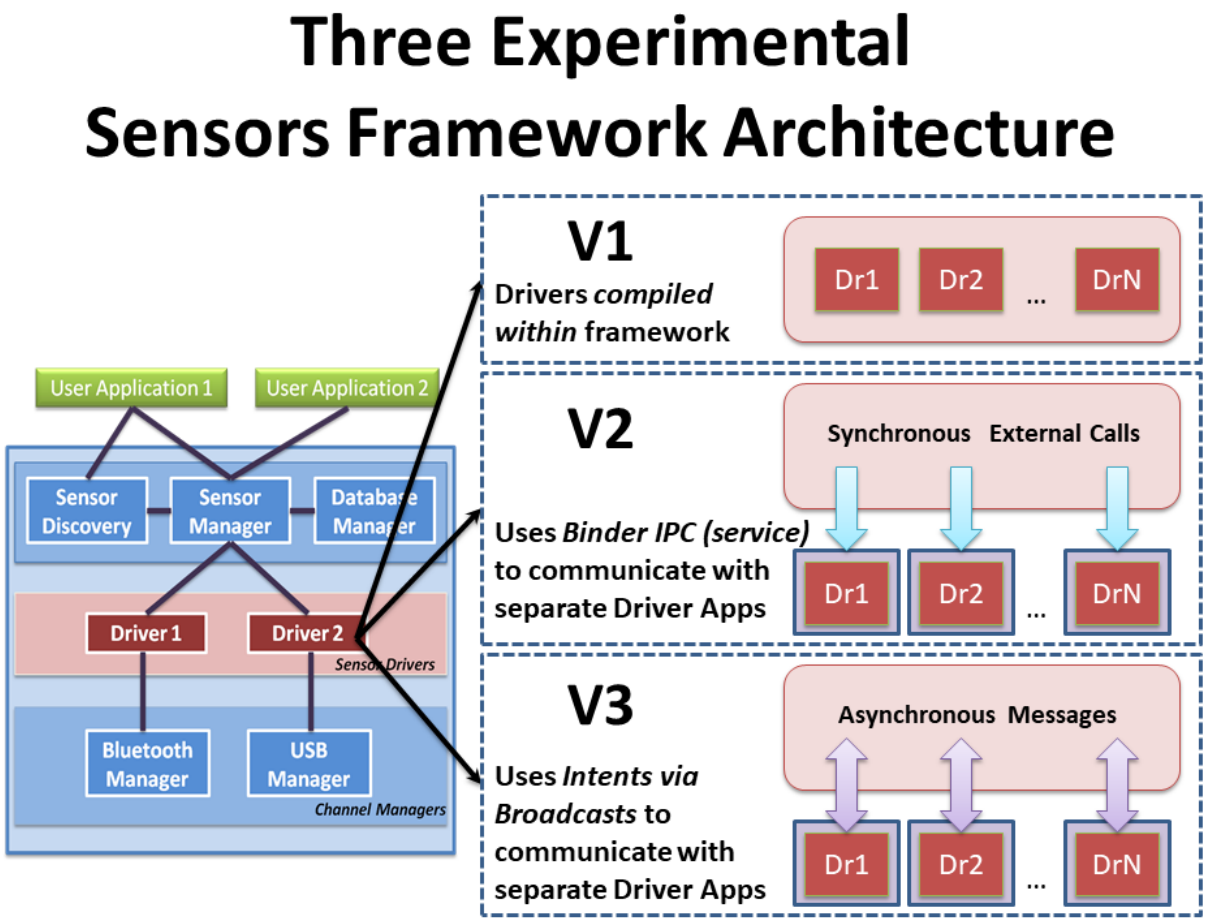



Figure 4.6: Architecture overview of the differences of the three *Sensors* framework implementations (V1, V2, V3). The different architecture implementations were used to evaluate the performance of different Android inter-process communication (IPC) constructs.

server. When the client calls these functions, the system copies the payload from the client into operating system memory, which is then moved into the frameworks protected memory. The framework then handles the processing with the data provided in memory. The sensor driver application includes driver-specific metadata that is used by the framework for driver discovery. 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 Service for inter-process communication. 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 sensors of the same sensor type it is important that it be stateless. Therefore, the driver's inter-process communication are designed to transfer the required state to the driver from the *Sensors* framework 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.

**V3: Driver Communication via Broadcasts** – The V3 version of the framework also creates separate Android apps for each sensor driver but uses message passing with Android Broadcast Receivers for inter-process communication. Each driver specifies a unique broadcast address that is discovered by the framework during the driver discovery phase (described in Section 4.4.4). 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 creating a Broadcast Receiver construct 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 sensors that are 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. One advantage of the V3 architecture is it decouples the sensor drivers from the thread of control allowing for a non-blocking message passing framework, but V3 incurs the

performance 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 Android Broadcasts can be intercepted by another Android application, which is a security risk. This could be addressed by encrypting data sent between the different processes.

*Verification of Example Deployment Applications*

A few example applications were implemented to demonstrate the usability, flexibility, and extensibility of the framework. These applications helped to evaluate whether the *Sensors* framework provided abstractions that were reusable by different use cases and verified the framework and sensor driver's interfaces. The example applications exercised 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:

- **Single Reading:** The *end-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 'mPnuemonia' use case (described in Sections 3.5.1 & 5.1.1 ) is an example of an application that connects sensors (e.g., blood pressure, pulse oximetry) to the mobile device. This use case has an approximate throughput of 1.5 packets/second over Bluetooth or USB since the *end-user* only needs to determine when the value has stabilized for recording the reading.
- **Real-Time Time-Series:** The *end-user* observes a stream of data values from the sensor in real-time that might need to be acted upon. Monitoring the temperature curve of the milk pasteurization procedure [27] is an example of this use case. The temperature curve can also be saved so that it can be reviewed later. The temperature values had an approximate throughput of 1 packets/second over USB for monitoring the milk pasteurization.

- **Snapshot Time-Series:** Sensing devices are deployed to autonomously monitor certain phenomena. The sensors aggregate readings internally over a period of time and may report some data from a remote location to a centralized data repository 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 [26]. The sensor readings over USB had an approximate throughput of 1.0 packets/second.

- **Historical Time-Series:** Sensing devices 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 establish a communication channel to offload the sensor's stored data. The WaterTime monitoring [25] application exemplifies this scenario. This scenario is also dependent on being able to configure the sensors in the field with sampling rates and identifiers since communication with sensors is only possible when within close proximity to the sensor. The uploading of cached sensor readings to the Android device had an approximate throughput of 51 packets/second over Bluetooth.

*Communication Channel Throughput*

To understand the constraints of the communication medium we tested the saturation point of the Bluetooth and USB communication channels. We performed a stress test that sent fake sensor data as fast as possible over the communication channels. We created a simple sensor that executes a loop for ten minutes that sent 1 byte data packets as fast as possible. The maximum throughput that could be achieved with full saturation of the communication channels using each of our three framework architectures is shown in Table 4.4.

Table 4.4: Framework throughput (packets/sec) on Bluetooth and USB Channels [19]

| *Framework* | *Bluetooth* | *USB* |
|---|---|---|
| V1 - Drivers in Framework | 58.0 | 42.5 |
| V2 - Drivers with Service | 58.7 | 41.5 |
| V3 - Drivers with Broadcasts | 49.0 | 40 |

*Power Usage*

The power consumption of each of the three frameworks was negligible when compared to the power used by an active Bluetooth connection or an illuminated screen. To test the power consumption we ran a 12-hour test where each framework processed 100 packets/second on a Nexus One. At the beginning of each test, the battery was charged to 100% and the screen, Bluetooth, Wi-Fi, and cellular radios were turned off. The only Android apps that were running during the test were the battery monitoring app and the *Sensors* framework app. Each test resulted in an approximate 3% drain of the battery during the 12-hour period. 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 mobile device components.

*Framework Throughput*

To evaluate the performance of each framework design, we created a fake communication channel for fake sensors to use with varying send rates and packet sizes. The 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 4.5). To understand the effects of multiple sensors, we ran tests that varied the number of sensors that communicated simultaneously through the framework (Table 4.6). 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 4.7). Understanding the implica-

tions of using Androids that have different costs is important because users in low-resource regions will likely have a diverse set of devices.

We measured framework throughput (packets/second) by sending data at varying rates with varying sizes using a Nexus One device. We varied the parameters to understand any limiting factors inherent to the different inter-process(IPC) mechanisms. The test results shown in Table 4.5 report on the number of packets received on average per second for each of the frameworks V1, V2, and V3. Each test ran for three minutes on five different Nexus One phones with the results of the tests averaged. The send delay values began at 1 millisecond and were increased by doubling the delay up to a maximum of 128 millisecond, 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. 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 1 millisecond delay because of memory errors invalidating the results (indicated by "Error" in the table). In the cases where the send delay was not the dominant factor, V1 appeared to perform the best since it 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 Android Broadcast Intent system would have more overhead than a synchronous Android Service 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[69]. While the double IPC call for V3 causes slower performance, the effects are generally small except when data is being generated at faster rates. Our results also matched the findings that there is a drop-off in performance for large packets. The Android system allocates a default operating system buffer 4 kilobytes in size for each process that will receive data from Android Service calls. When the payload size exceeds 4 kilobytes, the system allocates an additional, temporary buffer to transfer the data. These larger packets incurred additional allocation overhead and

Table 4.5: Throughput results (packets/sec) for a Nexus One when varying the packet size and inter-packet generation delay for the three framework versions [19]. 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 |
| **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 |

were shown to be inefficient because the Android operating system maps each additional memory page separately, instead of mapping them all at once [69]. These factors result in a drop-off in performance for payload sizes over 4 kilobytes, which is reflected in our results for packets 10 kilobytes 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 64 milliseconds. In this test, the drivers simply copy the incoming bytes into a parsed sensor reading to avoid any performance penalty from the normal processing of sensor data. The results in Table 4.6 show that the frameworks perform close to the theoretical maximum 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 the realistic applications for low-resource deployments that we have considered to date.

Table 4.6: Framework throughput (packets/sec) of multiple sensors sending 100 byte packets every 64ms on a Nexus One [19]

| **Number of Sensors** | **Max** | **V1** | **% of Max** | **V2** | **% of Max** | **V3** | **% of Max** |
|---|---|---|---|---|---|---|---|
| **6** | 93.8 | 93.3 | 99.5% | 93.2 | 99.4% | 93.1 | 99.3% |
| **12** | 187.5 | 186.5 | 99.5% | 186.4 | 99.4% | 185.6 | 99.0% |
| **24** | 375.0 | 371.9 | 99.2% | 371.8 | 99.1% | 365.8 | 97.5% |
| **48** | 750.0 | 737.6 | 98.3% | 735.2 | 98.0% | 686.4 | 91.5% |

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 experiments for multiple sensor performance, we used 100-byte packets sent every 64 milliseconds 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, con-

firming that the framework should be portable to a variety of Android devices. While the results presented in Table 4.7 show all of the devices were able to support multiple sensors with a send rate of 1 packet every 64 milliseconds, other tests showed that faster send rates eventually caused errors on all devices. Devices with smaller amounts of RAM and smaller default memory heap sizes seemed to have more issues. For instance, when delay between packets was reduced to 32 milliseconds 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 4.7: Throughput (packets/sec) of V2 framework on different Android devices using 100 byte packets that were sent every 64ms [19].

| **Number of 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 |

*Summary of Results*

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 three frameworks' 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 trade-off in terms of programming ease, deployment ease, and performance [19]. The separate driver app design makes it easier for *end-users* to add new drivers dynamically, 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.

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. A disadvantage of V2's design is its inter-process communication is 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 because any Android app can register to receive broadcasts making it easy for rogue apps 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 *Sensors* framework; however, during construction no communication between drivers and framework is possible until after construction is complete because no messages can 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.

### *4.4.4 Sensor and Driver Discovery*

The *Sensors* framework semi-automates the sensor discovery process by providing an existing user interface that 'Application Developers' can launch when they need the *end-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 with the desired sensor. The *end-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. The *Sensors* framework automatically discovers installed driver apps by searching Android manifest files. Once the *end-user* has mapped the sensor with the appropriate sensor driver the framework will automatically handle the processing of sensor data. After the *end-user* selects the sensor they want to use, the Sensor ID (SID) is returned to the application. This SID enables the application to control the sensor with the functions presented in the framework interface. Once a sensor has been discovered it is stored in the SensorManager database, enabling the application to skip the discovery step in the future.

An advantage of having the sensor device-drivers as separate Android apps is that it simplifies driver distribution. Ideally, we envision a scenario where manufacturers post their drivers on their own website or on an Android marketplace such as Google Play. By taking advantage of Google's standard Android app distribution system, the framework is designed to lower barriers for novice users by using familiar app delivery channels. Additionally, if the device manufacturer needs to update their device driver they simply need to post the updated app to the Android marketplace, since Android's built-in app update system will handle distributing the update. 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 proprietary drivers that can be easily integrated into the *Sensors* framework thereby protecting their proprietary protocols.

### *4.4.5 Related Work*

Three external sensing frameworks were influential in the design of *Sensors* framework's abstractions 1) the Reflex project that focused on proper abstractions to interact with low power processors to enable low-power operation of external sensors [87]; 2) Amarino made it easy to prototype sensors using an Arduino [6] over Bluetooth [76], and 3) Hijack connected external sensors to phones using audio jacks [80]. While these sensing frameworks were

influential in the development of *Sensors*, they were individual solutions that did not focus on the interoperability of all elements such as data collection, analysis, and visualization on the device. There is also a significant body of research in device driver design that examines trade-offs of reliability, ease of use, and performance with user-level versus kernel-level drivers or a combination of the two [51, 75, 82, 85, 118, 120].

The Reflex project [87] is a fork of Dandelion project. Dandelion envisions a scenario where sensor vendors provide a runtime to enable a platform-agnostic programming abstraction called a 'senselet,' that enables developers to write code that runs on the sensor itself. The *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 a developer in this limited sensor environment. Additionally, *Sensors* does not require sensor vendors to include a runtime, thus enabling the framework to support any standard sensor that communicates via a supported communication channel. Reflex 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 coprocessors. Dandelion [86] supports building applications distributed across a Maemo Linux smartphone and wireless body sensors by providing abstractions that shield 'Application Developers' from hardware specific code. While Reflex focuses on energy efficiency and performance in mobile-sensing applications, *Sensors* focuses on lowering programming barriers for 'Application Developers' and supporting different data and application types.

Like *Sensors*, the Amarino [76] project is a toolkit that connects Android devices with Arduino microcontrollers to create a sensing platform. Amarino focuses on connecting to the Ardunio wirelessly over Bluetooth as opposed to *Sensors* which uses a wired connection over USB. Another rapid prototyping platform that eases integration with embedded hardware

devices is Gadgeteer [145], which integrates modular hardware components with object-oriented programming in C#. While Gadgeteer and *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 *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.

The AndWellness [67] project lets researchers customize surveys to collect data from sensors on mobile devices carried by study participants. It shares *Sensors* framework's goal of lowering barriers for building sensing applications. However, unlike *Sensors*, AndWellness focuses primarily on customizing surveys and front-end visualizations of real-time data. *Sensors* aims to lower barriers for the 'Application Developer' and 'Sensor Driver Developer' to make it easier to leverage ODK 2 tool suite to enable organizations to build custom data management applications. PRISM [34], like *Sensors*, is sensing application middleware whose aim is to provide reusable components and eliminate redundant efforts. PRISM has been evaluated for applicability to a variety of applications, all of which interface exclusively with sensors built into the mobile device. In contrast, the *Sensors* framework adds abstractions that allows interactions with both built-in sensors and external sensors by abstracting away the communication layer to make a common sensors interface.

The concept of user-level drivers (or application-level) is not new, as Leslie et al. [82] 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 [51] 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, hardware input and output) and moving the rest of the driver code to a user-mode process. Similarly, Decaf Drivers [118] 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 *Sensors* was influenced by these related projects, it focuses on creating user-level drivers for locked

consumer devices running Android. Therefore, unlike these other projects, we do not alter the kernel to provide the communication link between the operating system and the user-level driver. Instead, *Sensors*' communication managers run as user-level threads and use Android inter-process communication 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 partially motivated by the desire to make systems more fault-tolerant and reliable in the face of device driver errors. Maverick [120] a web-based system, provides security by using device drivers and frameworks that run as user-level web applications to support interactions with multiple USB devices. Alternatively, Carburizer [74] detects and tolerates interrupt-related bugs to proactively manage device failures for improved reliability in the presence of faulty devices. Like Maverick, *Sensors* leverages user-level drivers to provide reliability and security; however, *Sensors* runs each driver as a separate Android app, causing each driver to be isolated in its own virtual machine (i.e. JVM).

### *4.4.6 Summary*

The *Sensors* framework helps ODK 2 meet one of its four design principles: 'new sensors, data input methods and data types should be easy to incorporate into the data collection pipeline by individuals with limited technical experience.' *Sensors* application-level device driver framework enables reuse of sensor-specific code between various applications by separating the high-level application logic from the underlying sensor device-driver logic. The framework is designed to be flexible in terms of data type, data collection rate, data size, sensor configuration requirements, and communication channel. By lowering the barriers to add external sensing components, the *Sensors* framework seeks to expand mobile data collection applications to include an even richer set of data types. *Sensors* framework helps to make it easier for *deployment architects* to create custom *workflows* and *dataflows*.

After testing the three framework implementations, it was clear that performance was not the most important factor to consider when selecting the final architecture. Based on our

requirements gathering, most sensing applications sample data at a significantly lower rate than the framework's maximum throughput. Additionally, the performance analysis revealed that the system bottleneck is the USB and Bluetooth communication channel throughput rather than the *Sensors* framework throughput.

By creating a framework designed to follow ODK's modular components philosophy, *Sensors* seeks to enable *deployment architects* to augment their Android device with external sensing options. The integration of data from a variety of sensors through a single interface for both wired and wireless communication channels simplifies application development, as the single interface abstraction reduces the amount of code needed to access a sensor. The modular framework 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 operating system. 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.

### *4.5 Submit*

Despite growing access to cellular coverage throughout the world, sufficient bandwidth to share the megabytes or gigabytes of data generated by data management applications is not always available, convenient, or cost efficient. Unfortunately, data is often transmitted using whatever protocol a software developer selected when implementing the software. Dynamic selection of available protocols based on a deployment's context and user location could improve connectivity in challenged network environments. *Deployment architects* often have to adapt *workflows* to the realities of resource-constrained contexts where issues such as affordability, infrastructural constraints, institutional capacity, and technical support are nontrivial. To address challenges faced by *deployment architects* trying to customize their

deployments, we created the *Submit* framework. *Submit*[23] focuses on creating data communication abstractions to enhance ODK's functionality to adjust to data importance, battery, cost, and local policy concerns in disconnected environments and areas of sparse heterogeneous connectivity.

*Submit* enables application-level communication optimization of sparse heterogeneous networks by sending appropriate data over available network infrastructures or peer-to-peer communications. The data transmission method selected should be based on an organization's usage model and priorities, as a one size fits all solution will not work for the variety of ICTD use cases. For example, forest workers do not have reliable access to electricity, potentially for weeks, making battery life the highest priority. Occasionally replicating data between devices in the field can reduce the chance of data loss. Data importance and other data characteristics set by a *deployment architect* should determine when and how much data should be replicated to conserve battery life. In contrast, community health workers have better access to electricity, but experience variation in the frequency in which they come into range of inexpensive connectivity (e.g., Wi-Fi at a clinic) and in how often they meet other workers in the field. As workers meet in remote areas, peer-to-peer technologies could be used to transfer data so that a worker with more clinic visits could act as a Data MULE [127] and transport information with low priority, reducing the amount of data that needs to be sent over costly cellular networks. *Submit* provides a framework for organizations to adapt their applications to share data using appropriate technologies for their various network conditions and data communication needs. The *Submit* framework focuses on accounting for inherent data characteristics, contextual data characteristics, and network characteristics when making decisions on what and when to transport.

*Submit's* design incorporates information from the multi-perspective design described in Section 1.1 and attempts to hone how and when information is transmitted in challenged networking environments. The framework combines information from the three perspectives to optimize data transmission in challenged network environments by utilizing multiple heterogeneous networks (e.g., cellular, Wi-Fi, peer-to-peer), based on contextual data char-

acteristics and available connectivity. Urgent data can be transmitted over more pervasive, higher cost networks, while less urgent data can be transmitted opportunistically over less pervasive, lower-cost networks. ODK deployments are common in rural environments with sometimes sparse connectivity. In a May-June 2015 online survey of the ODK community by Cobb and Sudar et al. [31], 48 out of 56 respondents who deploy mobile devices for data collection reported deployments in rural environments. Respondents from all environments reported that the total size of data collected varied considerably, with 19.6% of respondents collecting gigabytes, 53.6% of respondents collecting megabytes, 10.7% of respondents collecting kilobytes of data, and 16.1% of respondents did not know the size of their data collection. Transmitting gigabytes over expensive connections could be cost prohibitive, so understanding the importance of a piece of data is crucial when deciding to transmit over expensive connections.

To better facilitate dynamic selection, the traditional concept of the TCP/IP Application Layer [132] should be extended to include: 1) metadata from the platform about connectivity; 2) inherent data characteristics from the software/application developer; and 3) contextual constraints from a *deployment architect*. *Submit* helps to simplify mobile app development by managing connections, providing a user interface, and abstracting various transmission protocol libraries. For long periods in disconnected environments, *Submit* supports peer-to-peer data transfer using Wi-Fi Direct, Bluetooth, both versions of NFC available on Android, and QR Codes. It simplifies the process of leveraging Android's peer-to-peer networking capabilities by notifying *end-users* when to exchange data and includes abstractions that simplify peer-to-peer connection setup. *Deployment architects* can predetermine parameters for peer-to-peer communication, such as which transfer mode to use. *Submit* presents a unified peer-to-peer interface to the *end-user*, prompting them for missing information (e.g., select Bluetooth device to transmit).

### *4.5.1 System Design*

The *Submit* framework app uses an Android service to coordinate data communication by providing channel monitoring and transmission scheduling mechanisms to other Android apps. *Submit* provides software developers with an interface that abstracts communication channels and flexibly handles data ownership and application-specific synchronization issues. *Submit* is designed to separate application logic from the network routing logic with a communication system that provides extensibility in terms of: 1) adding transmission channels; 2) modifying transmission channel selection; and 3) handling complex data ownership and application-specific synchronization issues. *Submit's* service API exposes communication scheduling mechanisms that 'client apps' use to either 1) delegate responsibility of transmitting data to *Submit*; or 2) register to receive notifications when appropriate network channels are available. For the purposes of this section, the term 'client app' refers to any Android app that binds to *Submit*'s Android service. When using *Submit* for notification purposes, a client app takes responsibility to transmit its data with its own possibly proprietary or complex protocol.

The *Submit* framework enables developers to flexibly integrate their app-specific protocols with *Submit's* networking logic. Since many synchronization protocols are complex and possibly proprietary, it would be difficult to create a generic tool that conformed to all possible synchronization protocols. Instead, *Submit* only requires an app to provide contextual data characteristics and metadata (e.g., type, size, priority) and allows a mobile app to maintain ownership of transmission protocols. This means the choice to use *Submit* does not preclude the use of another synchronization protocol or encryption schemes as an app can leverage *Submit's* communication management for only a subset of its data.

To enable *deployment architects* to specify network parameters, *Submit* is designed to leverage ODK 2's XLSX format to enable the *deployment architect* to further define characteristics about data when customizing their data storage. *Deployment architects* already use the XLSX format to specify names of data fields, define the field's type, provide question

text, and specify flow logic. We explored the use of a basic set of parameters that *deployment architects* can input into the XLSX system in relative terms (1-10 scale), including data priority (how quickly the data should be transmitted) and data importance (how important it is that the data not get lost). Additionally, we investigated the creation of transmission deadlines in terms of time elapsed after the data was collected. To minimize the burden on the *deployment architect*, all parameters were kept optional and - if unspecified - automatically set to lowest values, causing the *Submit* framework to optimize for low-cost routing. Since network costs vary across geographies, additional information containing a cost model would need to be specified by an organization. This could be accomplished by specifying a configuration file with cost per byte or cost per message for each transmission medium.

Client apps specify two types of objects to interface with *Submit*: 1) the 'DataPropertiesObject' and 2) the 'SendObject.' The 'DataPropertiesObject' contains metadata that describe the data to be transmitted. The properties supplied include: data size, data urgency, data fragmentability, and reliability requirements. The inherent data characteristics are derived from the perspective of the developer. What represents "normal" sized data for the client app is an inherent property that can vary if the app primarily transmits answers to survey questions rather than an app that primarily transmits simple reminders. The answers to survey questions are not likely to fit in data space available in a single SMS message, whereas a simple reminder might be able to fit within a single SMS message. By obtaining the software app's perspective on data size, *Submit* can calibrate its routing mechanisms to best handle the common communication case on a per app basis by narrowing the selection of appropriate channels. The 'SendObject' contains a list of 'Destination Addresses' that define the type of transport as well as the necessary parameters to use for the transport. For example, to utilize HTTP POST, the 'Destination Address' would contain a URL; to utilize an SMS channel the 'Destination Address' would include a phone number. By implementing 'Destination Address' as an abstract type, *Submit* is extensible to various communication protocols. 'SendObjects' also contain the file path to, or string representation of, the data to be sent on behalf of the client app.

Shared data between *Submit* and client apps could create "race conditions," as apps are often dependent on their internal data stores being correct and consistent. *Submit* addresses ownership issues by removing ambiguity through the assumption that the client app owns the data until it explicitly grants *Submit* temporary ownership rights when it delegates transmission responsibility to *Submit*. If an app only provides a 'DataPropertiesObject' which has no pointers to the data, *Submit* assumes the app is maintaining ownership of the data, as the client app is only asking for a notification of when to send the data. In contrast, if the client app provides a 'SendObject' containing the actual data or a pointer to an accessible external file, *Submit* retains ownership of delegated external data until *Submit* notifies the client app that it no longer needs ownership of the data by providing the final status of the data transmission. Since both the client app and *Submit* can be responsible for sending data, they must communicate the success or failure of data transmission. Broadcast intents are used to synchronize the sending status. When *Submit* is responsible for transmitting the data, it broadcasts the status of the communication exchange to the client app. Likewise, if a client app has been notified that it is the appropriate time to send the scheduled data, the client app broadcasts the transmission result status to update *Submit*'s internal state.

If the client app delegates sending-responsibility to *Submit*, the 'SendManager' selects an appropriate network to transmit the data based on the 'Destination Addresses' and whether there is a transmission protocol for the available network. *Submit's* 'CommunicationManager' is responsible for determining whether an available channel is appropriate for submitted data. The 'CommunicationManager' gauges an available channel's bandwidth capacity and costs. The 'ChannelMonitor' listens to system broadcasts for changes in connectivity, including Wi-Fi events, ad hoc communication opportunities, and cellular events. It reports back the current state of connectivity to the 'SubmitService' when a change is detected. The 'SubmitQueueManager' iterates continually over the pending data that needs to be transmitted (described by 'DataPropertyObjects'). With each pass through the queue, it updates the state of each pending data item based on the results from *Submit's* protocol modules.

Table 4.8: Average latency for a client app sending data using *Submit* and without using *Submit* [23]

| | Wi-Fi w/ Submit | Wi-Fi w/o Submit | 3G w/ Submit | 3G w/o Submit |
|---|---|---|---|---|
| 10 KB | 0.12s | 0.10s | 0.59s | 0.47s |
| 100 KB | 0.40s | 0.28s | 1.66s | 1.28s |
| 1 MB | 2.53s | 2.03s | 10.93s | 7.86s |
| 10 MB | 22.55s | 20.58s | 83.95s | 80.35s |

### *4.5.2 Experiments*

*Splitting Data Transmission*

To measure the baseline impact *Submit* has on a client app's communication performance, we integrated a simple file upload app with the *Submit* framework to evaluate network usage and latency. The test involved the client app using HTTP POST to upload data from a client to a server with and without *Submit*. Performance was measured for two scenarios: Wi-Fi only and 3G only. The tests were performed on a Samsung Galaxy running Android 4.3. The results in Table 4.8 show the average latency for each file size for the ten uploads. As expected, there is slight latency overhead associated with *Submit* due to its use of remote procedure calls and broadcast intents to communicate with a client app for each uploaded file.

*Submit's* latency additions are counterbalanced by its ability to minimize network usage according to *deployment architect* preferences. To verify the reduction of cost for network usage, a small experiment was performed for ten minutes where Wi-Fi was disabled, leaving only 3G accessible. In this experiment, the client app uploaded randomly selected files from a directory of files that ranged between 5 kilobytes and 100 kilobytes in size. For the client using *Submit*, a threshold value was set that prevented any file over 7 kilobytes from being sent over a mobile broadband network. After ten minutes, the Wi-Fi was re-enabled for 10 minutes. After the entire 20 minutes the number of packets sent over Wi-Fi was compared to the more costly 3G network. The client without *Submit* sent 380.6 kilobytes over 3G

Table 4.9: Possible reduction of transmission sizes if record is split by data type or data priority for the site visits scenario [23].

| | Bytes | Percent |
|---|---|---|
| Avg Total Record Transmission Size | 330,773 | 100.00 |
| Avg Data Size | 1,213 | 0.37 |
| Avg Photo Size | 329,217 | 99.53 |
| Avg Priority Data Size | 343 | 0.10 |

whereas the client using *Submit* only sent 12.8 kilobytes over 3G. Thus, *Submit* selectively uses one network while waiting for a cheaper communication channel to become available.

To understand *Submit's* effects on deployment scenarios, we used data from real deployments that highlight networking challenges and the benefits of leveraging contextual data characteristics when making data transmission decisions. The Government of Punjab's Health Department (Pakistan) used ODK to document the workload, staff attendance, and available medical supplies at health clinics. To identify inventory shortages it dispatched inspectors to verify inventory and photograph workers for attendance verification. This information was transmitted to ODK servers; however, only the medical supply data was time-sensitive. The photographic verification of attendance was a human resource concern and did not require urgent delivery, yet this non-urgent portion of data dominated the transmission cost because the large size of photos. To test the *Submit* framework, we used a small sampling of 85 actual site visit records to calculate data transmission reductions for this site visit scenario. Table 4.9 shows the calculated average reduction if *Submit* managed ODK Collect's data submission process. By simply separating transmission of the text portion of the data and delaying the transmission of the photo's binary data until the device is in free Wi-Fi range would mean that only 0.37% of the total data would be transmitted via cellular and 99.53% of the data would be transmitted over Wi-Fi. Using *Submit*'s concept of data priority could further reduce cellar transmission to 0.1% of total bytes by only transmitting the information about medication inventory.

*Peer-To-Peer Transmission*

We evaluated the performance of peer-to-peer transmission methods by comparing 5 methods of transferring data between Nexus 7 devices positioned ˜0.5 meters apart running Android OS version 4.4.4. Wi-Fi Direct and Bluetooth were tested using transfer sizes of 1 kilobyte, 10 kilobytes, 100 kilobytes, 1 megabyte, 10 megabytes, 100 megabytes, and 1 GB of data. NFC transfer sizes were limited to data transfer times that were feasible for generic peer-to-peer use (large transfers took too long). NFC with Bluetooth was tested up to 10 megabytes, while NFC-only was tested up to 100 kilobytes. Additionally, peer-to-peer transfer using QR codes was tested by having one Nexus 7 device display QR Codes on its screen while the other Nexus 7 read the QR codes with its built-in camera from ˜0.3 meters. The ZXing library was used to generate and scan the QR Codes. While QR Codes specifications state transmission up to 4 kilobytes of data [112] are feasible, our results show that transfers are unreliable past 1 kilobyte of data. The time it took to scan a QR Code was fairly consistent as the size of data increased, but the error rate increased to over 60% for file sizes larger than 1 kilobyte.

Bluetooth and NFC were the fastest transfer options for smaller amounts of data, as shown in Figure 4.7. As the data size increases, Wi-Fi Direct emerges as the fastest mode of transfer. Until data sizes exceed 100 megabytes, the total time for Wi-Fi Direct remains essentially constant because establishing the connection dominates the transfer time [24], as shown in Figure 4.8. Wi-Fi Direct is a realistic choice for data on the order of 1 megabyte or larger, but for anything lower than 1 megabyte, Bluetooth may be a better option. Figure 4.7 also shows that NFC-only is significantly slower than Bluetooth-enabled NFC[1]. QR scanning had the largest variance in the duration of file transfer. While increasing the error correction level of the code can help remedy this issue, for data sizes close to 1 kilobyte the QR Code was too dense to accurately and consistently be scanned.

Since battery life is important in disconnected environments, we evaluated the power

[1]http://developer.android.com/training/beam-files/

performance of Wi-Fi Direct, Bluetooth, and QR codes. For each test a connection was established between two fully charged Nexus 7's and data was continuously transmitted from one device to the other until the sending device's battery was depleted. For consistency, the device screens remained on at all times since the QR method requires the screen to be on and active usage of devices would cause the screen to be active some percentage of the time. The experiments revealed that despite being able to leave the device in airplane mode, the QR Code scanner consumed more battery charge than traditional data transfer methods. The QR scanner took 6.8 hours to drain the battery to a 10% level while Wi-Fi Direct transfer only lasted 0.33 hours longer. In comparison it took about 9.3 hours of Bluetooth transmission to drain the battery to a 10% level. The results suggest that the power required to continually use the camera and process bar codes resulted in greater battery consumption over time than Wi-Fi or Bluetooth transmission.

The main factors for selecting a peer-to-peer method include transfer time and battery efficiency. Per byte, Wi-Fi is more battery efficient, but within the data size range of 100 kilobytes to 1 megabyte, Bluetooth is faster. In the case of forest inventory workers, opportunity to charge the devices is the limiting factor and Wi-Fi should probably be selected, since it is more efficient per byte. For field workers, avoiding the slightly more cumbersome connection process of Wi-Fi might be more important. The main disadvantage of both Bluetooth and Wi-Fi Direct is the difficulty for the user to confirm which device they connected to. While this may be less of an issue in a forest inventory setting, it is one of the primary concerns in a clinical setting. With both Bluetooth and Wi-Fi Direct, someone attempting to steal data can spoof their device name and MAC address, potentially deceiving a user. NFC and QR Code communication allows users to visually clarify that the correct device is receiving the data. This can be an important advantage when the information is confidential, such as medical data. The results show that the QR Code scanner is slower and less power efficient than NFC with Bluetooth. However, there is the possibility of hand-to-hand contact when using NFC during the moment when the two devices are brought close together to establish the connection. Hand contact could be a disadvantage in a remote clinical setting where

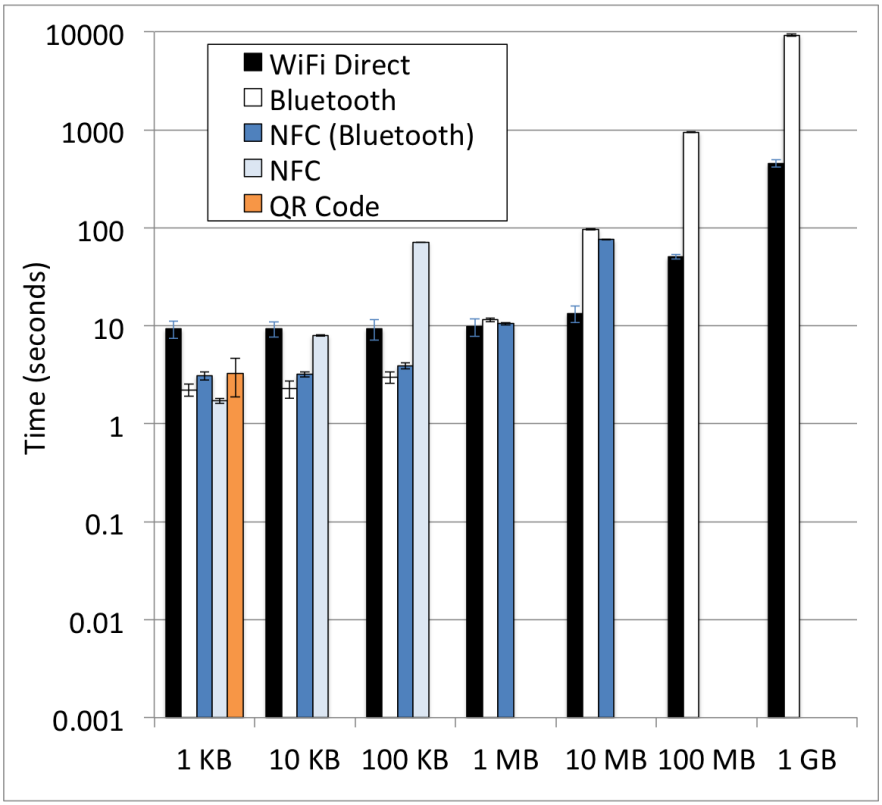



Figure 4.7: Data transfer times associated with peer-to-peer technologies with different file sizes (Log Scale) [23].

hygiene practices might restrict such contact. A key advantage of Bluetooth over Wi-Fi Direct is its ability to pair devices ahead of time, allowing users to more confidently send their data to the correct person. However, if NFC is not an acceptable option because of possible hand-to-hand contact or data size, white-listing Bluetooth devices could increase security in a clinical setting.

*Usability of Peer-To-Peer Transfer*

To understand how user interaction requirements would affect the time it takes to transfer data, we conducted basic usability tests with 22 participants that applied each of the different peer-to-peer data transfer modalities. The participants' ages ranged from 18 to 56, with a mean age of 25. After initial demographic information was collected, participants were

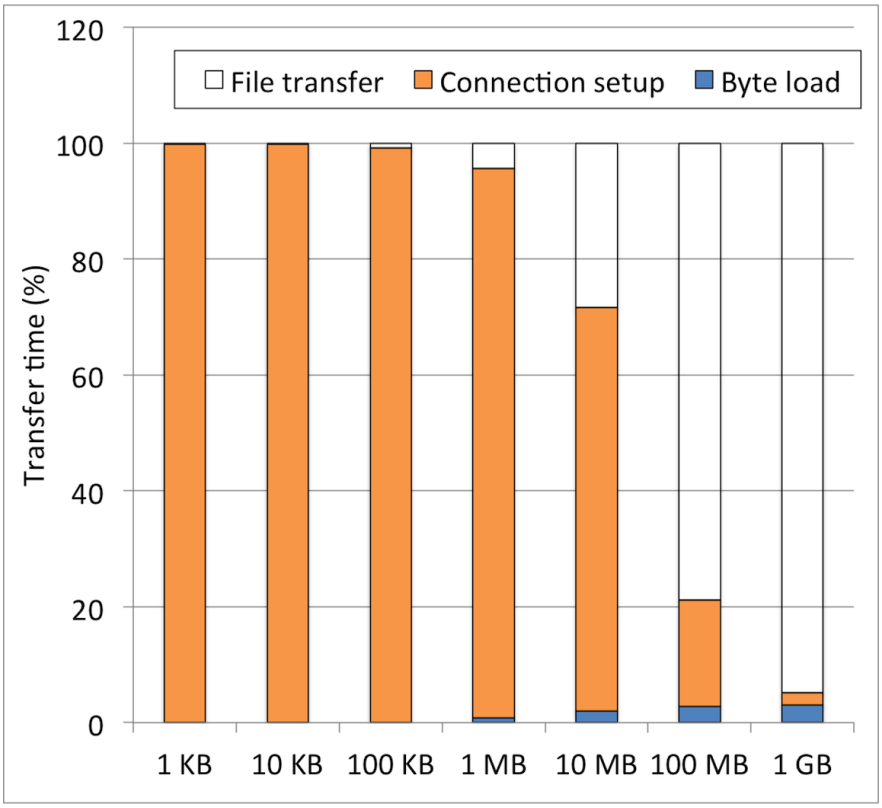



Figure 4.8: Percent of time spent in different phases of Wi-Fi Direct transfer. Connection setup time dominates small file size transfer [23].

given a short training session on how to use *Submit's* manual peer-to-peer transfer screen. Participants were then given a list of ten 1 kilobyte transfers tasks to complete, one sending and one receiving for each of the five transfer methods. The order of the task list was randomized across participants so that each transfer method appeared with similar frequency at each position. The first two tasks for each user used a specific transfer mode (e.g., send using NFC, then receive using NFC). After the first two tasks, participants were asked to complete the NASA TLX[2] form to rate the difficulty of the specific transfer mode. Once this was completed, participants proceeded with the eight remaining tasks. After the ten tasks were completed, a semi-structured interview was used to solicit feedback about the most confusing part of the transfer process and to help identify possible improvements to *Submit's*

[2]http://humansystems.arc.nasa.gov/groups/tlx/

peer-to-peer transfer. Participants were also asked to rank the five transfer methods based on ease of use (on a rank scale from 1 to 5, where 1 was the easiest transfer method to use and 5 was the most difficult), and rank them based on efficiency (on a rank scale from 1 to 5, where 1 was quickest transfer method and 5 was the slowest method).

These usability results confirmed the results from transmission performance. Users found that using a QR Code to transfer data was both the least efficient ($p < 0.001$) and the most difficult method ($p < 0.001$) to use[3] with a mean efficiency rank of 4.7 and a mean difficulty rank of 4.6. In three cases, users were unable to successfully scan the QR Code due to the lighting conditions of the room. Users also found that Bluetooth and Wi-Fi Direct were the fastest ($p < 0.001$) and easiest ($p < 0.001$) to use. Wi-Fi Direct had a mean efficiency rank of 2.3 and a mean difficulty rank of 2.3. Bluetooth had a mean efficiency rank of 1.9 and a mean difficulty rank of 1.8. This differs slightly from the data gathered during channel testing. Wi-Fi Direct was expected to be outperformed by all four of the other transfer methods because of the longer connection setup time. This discrepancy can be explained by the fact that users had to select more information during the user testing. The time it took to select all this information caused the actual speed of transfer to matter less when considering the overall time spent using the application. Additionally, the time it took to move the devices together for NFC and the time it took to line up the devices for QR Code scanning was not accounted for in the evaluation of performance. The usability issues found for NFC and QR Code dramatically increase the overall time it takes to transfer data. Testing results also showed QR Codes take an unreliable amount of time to transfer data because of environment conditions such as reflections due to lighting. However, based on the data collected from the NASA TLX form, there was no significant difference between the average level or type of stress users experienced while using the different transfer modes ($p = 0.57$)[4]. This implies that while users do prefer some transfer methods above others, the difference between the methods is relatively small compared to the overall ease.

---

[3]Significance calculated using the Kolmogorov-Smirnov Test.

[4]Significance calculated using the Kruskal-Wallis Test.

While users found Bluetooth and Wi-Fi Direct to be the fastest and easiest methods and QR Codes to be the slowest and most cumbersome, results were more varied for the NFC options. Some users liked the strong visual and physical cues associated with holding the devices together and touching the screen to beam the data between devices. Users also appreciated that the receiving NFC user does not have to select any options since the parameters, such as from whom you are receiving, are all automatically inferred when you hold the devices together. Other users felt uncomfortable with the inevitability of hand-to-hand contact that comes from holding the devices together with NFC. Other users expressed concern of potentially dropping the device. Most users held the tablets together with one hand and tapped the screen with their other hand. They thought NFC was slightly more inconvenient than other mediums, but would not mind using it.

Which peer-to-peer transfer method to use should also be based on inherent data characteristics (e.g., data size), contextual data characteristics (e.g., data importance, security), and battery constraints. Our experiments showed the most significant barrier to peer-to-peer transmission is the time it takes for users to set up a peer-to-peer connection. To address these challenges, Submit handles peer-to-peer connections internally and automatically populates most settings before providing a unified user interface.

### *4.5.3 Related Work*

Similar to *Submit*, other research has explored creating frameworks to leverage multiple networks for data transmission [9] that can split data over multiple networks based on cost and availability [10, 59, 84]. While work exploring simultaneous data transmission over multiple interfaces has been shown to improve mobility, power efficiency, and network capacity [9], our work focuses on providing configuration primitives for selecting a single network from a heterogeneous combination of different transmission opportunities in order to enable organizations to customize their transmissions to the deployment context. The *Submit* framework is most similar to other research work that identifies the best type of network for data transmission given a configurable policy and contextual information. MultiNets [99]

proposes real-time switching on mobile phones between different network interfaces using policies based on power, throughput, data offloading,and latency; however, the policies are applied to every app on the device. *Submit* provides a library that allows a *deployment architect* to configure policies that will only be applied to apps relevant to the application. In this way, *Submit* is similar to Delphi [38], a transport layer module that selects the most appropriate network for data transmission given policies set by applications. However, Delphi assumes operation in an environment with ubiquitous connectivity and focuses on the transport layer rather than the application layer meaning that it does not address many of the issues faced in resource-constrained environments such as intermittent connectivity and variable pricing policies. Also, in contrast to Delphi, *Submit* uses information about data (e.g., time-sensitivity, importance) to identify the best method for transfer. Similar to *Submit*, Haggle [126] separates application logic from compile-time determined transport bindings so that applications can be communicative in dynamic networking environments. Haggle and *Submit* both provide API's to developers, but *Submit's* design goes further and provides configuration constructs for *deployment architects* to adjust their mobile information system. Another issue with Haggle is that it proposes a general form of a naming notation to allow for late-binding that is independent of the lower-level address. While being flexible with late-binding addressing is a good idea, the infrastructure is not currently available to support these assumptions, which is problematic for organizations operating in diverse environments with limited infrastructure. Our work distinguishes itself from previous research in that it seeks to provide a network management module that is configurable at the application layer in order to enable flexible control to a humanitarian organization operating in various conditions, including in areas of limited connectivity.

Graasp (also known as Graspero) [146] is a social media web platform that was designed for internal knowledge sharing within the Doctors Without Borders (DWB) organization. Since DWB often operates in network-challenged environments, Vozniuk et al. investigated several types of existing solutions before developing their own peer-to-peer middleware content delivery solution called GraaspBox [147]. The first type of solution was mechanical

delivery using physical media, such as a USB stick or a hard drive. The second type of solution utilizes an available network that can have poor quality through the use of intermediate servers. This second type of solution includes projects such as TroTro [83] and Kwaabana [73]. Vozniuk et al. also point out that content delivery networks [125] are similar to the second type of solution, as they try to offset network quality issues with location proximity. The third type of solution applies various techniques on the user's device to either cache for asynchronous download or use peer-to-peer intermediaries. Two examples of device network optimizations are COCO [128] and *Submit* [23]. In GraaspBox's [147] discussion and conclusion section Vozniuk et al. report on issues developing the system, such as peer-to-peer complexities and conflict resolution that we also experience developing *Submit*.

### *4.5.4 Summary*

The *Submit* framework was an exploration into how to better enable *deployment architects* to customize their *workflows* and *dataflows*. *Submit's* abstractions decouple data and connectivity to enable application-level optimization of sparse heterogeneous networks. *Submit* supports customization of *dataflows* by not treating data uniformly; instead, individual pieces of data can be systematically assigned different communication parameters based on the deployment context. *Submit* aims to empower *deployment architects* to shape the communication priorities by optimizing a communication transfer method based on inherent data characteristics (e.g., data size), contextual data characteristics (e.g., data importance, security), and battery constraints. Based on these characteristics, *Submit* enables customization of *dataflows* that allow data to be transferred and shared using various networking technologies, including peer-to-peer transmission. Our experiments showed the most significant barrier to peer-to-peer transmission is the time it takes for users to set up a peer-to-peer connection. To address these challenges, *Submit* handles peer-to-peer connections internally and automatically populates most settings before providing a unified user interface to the *end-user*.

# Chapter 5

# ODK 2 DEPLOYMENT EXPERIENCES

To demonstrate the viability of the multi-framework approach, we engaged in multiple ODK 2 pilot deployments with various stakeholders to verify ODK 2's capabilities for broad use across subject domains. This chapter describes deployments that validate the suitability of ODK 2 frameworks to various use cases. By listening to our partners and understanding their needs, ODK 2 addressed some key limitations of ODK 1. Field experience with real organizations helps researchers understand the problems that prevent adoption of mobile data management solutions. By working with organizations to build real systems that are deployed in regions with varying constraints, we were able to test both ODK 1 and ODK 2 in various contexts with different infrastructures. Deploying mobile data management systems helps reveal system design assumptions that limit functionality because of expectations about deployment contexts.

ODK 2 has been used by more than 10,000 unique individuals as of August 2020, according to Google Firebase. It should be noted that this number is a minimum estimate of users, as it does not include users that use branded forks that are not tracked by Google. The Google Firebase statistics only track the posted generic ODK 2 mobile apps when they are first opened. ODK 2 has been piloted in 10+ subject domains and has been deployed in over 120 countries. ODK 2 has been successfully adapted by non-programmers to local contexts to satisfy organizational requirements and custom *workflows*. In this chapter, we discuss pilot deployments, ODK 2 usage, ODK 2 extensions to demonstrate the variability of use cases, ongoing field evaluations, and design iterations to the frameworks.

## *5.1 Case Study Deployment Experiences*

Section 3.5 presented case studies that were used to helped refine the requirements for ODK 2 frameworks. This section discusses the experiences from creating and deploying three of the six case studies: 'Childhood Pneumonia,' 'HIV Clinical Trial,' and 'Mosquito Infection Tracking.' These three case study deployments required different levels of support by the research team ranging from minimal assistance beyond general question answering to helping design the *workflow*. Through the case study implementations, we verified that ODK 2 frameworks satisfy the requirements listed in Chapter 3. Deployment experiences from the 'Disaster Response' case study is discussed in Section 5.3, while the 'Tuberculosis Patient Records' case study is discussed in Section 5.2.1.

### *5.1.1 Childhood Pneumonia*

There is a critical shortage of trained healthcare workers in developing regions; however, the availability of mobile devices provides a platform to assist lightly trained healthcare workers in performing complex medical screening tasks. As discussed in Section 2.3, previous projects found that digitizing the World Health Organization's Integrated Management of Childhood Illness (IMCI) [101] on personal data assistants (PDAs) led to increased medical protocol adherence while maintaining examination time [42, 93]. Unfortunately, many organizations struggled to implement IMCI's complex medical *workflow* using ODK 1 XForms.

To verify ODK 2's ability to implement complex medical *workflows* with support for adding arbitrary sensors, we partnered with PATH [105] (a non-government global health organization) to develop an IMCI-based [101] medical assessment tool to support health workers in low-resource environments. The project aimed to provide a mobile diagnosis and treatment application that assists lightly trained healthcare workers when screening for diseases. The medical tool was named 'mPneumonia' since the primary goal was to improve health care providers' accuracy in the diagnosis and management of childhood pneumonia, as childhood pneumonia is a leading cause of disease deaths for children under five [54]. The

enhanced mobile device integrates a digital version of the IMCI protocol with two Android apps for assessing the respiratory rate and oxygen saturation. The two primary ODK 2 frameworks involved in this project were *Sensors*, which handles the interface between the pulse-oxygen sensor and the mobile device, and *Survey*, which manages the IMCI *workflow*. The secondary goal of the project was to verify that ODK 2 tools were capable of building complex clinical applications. To evaluate the success in limited-resource settings, PATH established partnerships in Ghana and India to conduct usability, feasibility, and acceptability evaluation of mPneumonia.

A global health intern at PATH (i.e., a non-programmer) was able to digitize the IMCI protocol using *Survey's* XLSX *workflow* description format. It is important to note that the first working prototype of mPneumonia was produced by a global health intern, with no programming background, being the *deployment architect*. Based on feedback from user studies, additional customizations of the layout, styling, and presentation of prompts were made by an undergraduate computer science student using HTML and CSS since the intern had no web or programming experience. The framework's design of using web configuration files for presentation limited the amount of knowledge the undergraduate student needed to make changes. The design also isolated the requested modifications to a few files, providing a rapid iterative design process between PATH and the undergraduate student. The *Sensors* framework was used to integrate a USB sensor to record a patient's oxygen saturation. To evaluate usability, feasibility, and acceptability, PATH conducted interviews of 30 health care providers, eight health administrators, and 30 caregivers in Ghana. PATH found that mPneumonia had the potential to improve patient care and was feasible to integrate into a rural healthcare worker's workflow. Interviewees reported that mPneumonia was "easy to use" and gave confidence to their diagnosis and treatments [55].

### *5.1.2 HIV Clinical Trial*

Adaptive Strategies for Preventing and Treating Lapses of Retention in Care (AdaPT-R) was a five-year randomized control trial by the University of California, San Francisco. The AdaPT-R program deployed an ODK 2 based mobile patient tracking application in five medical clinics in Kenya. The clinics serviced approximately 65,000 patients, with about 17,000 of the patients being HIV infected. The goal of the AdaPT-R study was to determine why some HIV patients quit following up with the HIV treatment and prevention programs. Retaining HIV-infected patients in treatment programs is an essential aspect of a successful HIV response in Africa. Instead of using a singular approach for patient retention, AdAPT-R used multiple different retention approaches to study which approach or which combination of approaches produced the best patient retention for 1,745 HIV patients. The objective was to determine a strategy to optimize patient retention within resource constraints.

The AdaPT-R study had 18 employees using 17 Android devices to interact with patients. Research assistants were provided 1-2 hours of initial training and received two ˜30 minute refresher training sessions a year. AdaPT-R's data manager in Kenya also provided additional one-on-one training on an as-needed basis. Examples of one-on-one training include a 30-60 minute session to familiarize site leads with the ODK 2 software update process and a ˜20-30 minute session to teach *end-users* how to send compressed logs from devices to a manager for troubleshooting.

Early in the study, field workers experienced issues such as periodic application freezes because of database locking issues and sluggish navigation. The AdaPT-R workflow required users to move back and forth between *Tables* and *Survey*, thereby increasing the probability of database locking due to Android life cycle issues. The revised ODK 2 service-oriented architecture (discussed in Section 4.3) was built in response to some of the problems experienced by the AdaPT-R study. The AdaPT-R team deployed ODK 2's revised framework to one clinic initially. After experiencing immediate positive results, the AdaPT-R team decided to expedite the upgrade of the remaining clinics.

Having all study participants available on the mobile device let the AdaPT-R team move devices between clinics without reconfiguring the devices (or physically moving paper files). This was helpful when devices broke or when the AdaPT-R team needed to re-allocate a device to another clinic if one site had a higher volume of data entry compared to another. Additionally, AdaPT-R has research assistants who cover tasks for more than one clinic; having the data for multiple locations on a single device lets these employees use the same device for data entry at all locations.

When asked why they chose to use ODK 2, the co-primary investigator of the randomized control trial noted: *"We needed a solution for capturing data from multiple forms and that would allow longitudinal follow-up of individual patients. We had experience with earlier versions of ODK, so the new features of ODK 2 made it the only option for us if we wanted phone-based longitudinal form completion. Would definitely recommend ODK 2!"*

### *5.1.3 Mosquito Infection Tracking*

The World Mosquito Program (WMP) is an international research collaboration focused on helping to protect people from mosquito-borne diseases by introducing the naturally occurring Wolbachia bacteria into local mosquito populations. WMP is a non-profit operating in 12 countries to reduce the spread of Dengue, Chikungunya, and Zika. The program's goal is to protect 100 million people from mosquito-borne diseases by 2023. To help accomplish this goal, the WMP has developed a mobile app built on the *Tables* framework to provide workers with an interactive map of site locations, historical data about each site, individualized instructions on what tasks to perform at which sites, and a data capture application [20].

A key component of the WMP is regularly visiting multiple sites over a period of weeks and months. These visits require historical data to be available to field workers, who may not be the only workers visiting the same site. Their operational workflow requires ODK 2's bidirectional synchronization protocol to provide historical as well as timely data to their field workers during site visits to remote environments. This requirement was the driving factor in the WMP team's decision to use ODK 2 over ODK 1. When describing ODK 2's

benefits, they said, *"we needed two-way synchronization"* and that all other benefits were discovered and leveraged later.

WMP leveraged ODK 2 frameworks to handle system tasks and simply hired web developers to customize the user interface and workflow using *Tables* and *Services* framework abstractions. The WMP team designed and developed their own custom *Tables* application using consultants specializing in web development. They decided to use *Tables* because *"developers don't like to be restricted,"* and *Tables* gave them the flexibility to customize their solution to their needs without being in *Survey's* question-rendering framework. Furthermore, *Tables* let them *"focus on web dev"* rather than worry about editing Android code, as would have been necessary with an ODK 1 solution. An example customization was WMP's eschewing the provided Google Maps library for a runtime library called Leaflet.js [81] to serve their own map files from within ODK 2's frameworks through *Services'* web server. This enabled WMP to provide more accurate and customized maps that were stored on the devices to support workers in locations that lack Internet access.

The WMP started deploying the ODK 2 based mobile app in 2016, and at the time of this dissertation, continue to increase their usage and deploy ODK 2 to more countries. The program manager reports *"the app is scaling quite well"* and that ODK 2 apps are *"quite easy to use and we haven't had any acceptance issues."* The largest challenge to the team has been logistical, as the velocity with which the WMP team was developing and deploying required frequent changes to the data-model. Neither ODK 1 nor ODK 2 is well suited to this type of change. The WMP team is using JSON objects in the database to work around the issue, but in general, changing data-models remains an open problem in ODK 2. The design tradeoffs between document-based storage versus row-based storage are further discussed in Section 6.2.2.

## *5.2 Extensibility and Modularity*

ODK 2 tool suite focuses on creating frameworks that are extensible, scalable, modular, and adaptable to multiple use cases in a variety of low-resource locations. Through the

generalized application frameworks, ODK 2 allows users of varying technical skill levels to create custom solutions that support a wide range of scenarios. ODK 2 aims to create base programming interfaces, open standards, and free reusable libraries to minimize the start-up costs for organizations, researchers, and companies seeking to participate in the space. To create an ecosystem, ODK uses the permissive Apache 2 open-source license to enable free use and encapsulation of the technology. This allows companies to have business models based around both creating products with additional features and/or providing consulting assistance to global aid workers. This section discusses examples of people leveraging ODK 2 frameworks' existing functionality.

### *5.2.1 Scan*

*Scan* is a paper digitization framework that bridges the gap between paper and digital data collection without forcing organizations to wholly adopt either scheme [36, 37]. *Scan* is outside the scope of my dissertation, but it is part of the ODK 2 tool suite. Scan demonstrates the extensibility and modularity of the ODK 2 tool suite as both *Services* and *Survey* provide key functionality to *Scan*. *Services* provides *Scan*'s database and synchronization functionality. *Survey* provides the question rendering framework for verification of digitized data, as well as the user interface that *Scan*'s workflow uses to display segmented pictures of free-form text for the user to manually input.

*Scan* adds the ability to quickly and easily digitize paper forms in disconnected environments to the ODK 2 tool suite. Paper forms are adapted by adding *Scan*-compatible data input components, which include QR codes, multiple-choice bubbles and checkboxes, structured handwritten number boxes, and free-form handwritten text boxes. Both digital and physical versions of this form are generated: 1) a *Survey* form definition and 2) a *Scan*-compatible form that is rendered as a JPEG image file to allow organizations to print and distribute a paper version of the form to field workers. *Scan* digitizes completed paper forms in the field by taking a picture with the Android's camera. Computer vision algorithms that run locally on the device process the picture into image snippets corresponding to the

individual data components [36, 37]. These snippets (excluding free-form, handwritten text boxes that *Scan* cannot digitize programmatically) are then fed into other computer vision algorithms that digitize the respective values. The digitized values can be reviewed and free-form, handwritten text values can be transcribed by a data collector by viewing the form instance in *Survey* or *Tables*. The image processing is performed locally on the device without the need for an Internet connection.

*Tuberculosis Patient Records*

After Dell et al.'s work with *Scan* [36, 37], it became part of the ODK 2 tool suite. To verify *Scan's* capabilities and integration into the other ODK 2 frameworks, our research team partnered with Mercy Corps [91] and VillageReach [143] to pilot ODK 2's effectiveness at digitizing and validating paper patient records and providing basic reporting to clinicians in the field. This pilot program ran for three months, with four field workers working with 122 patients in two districts in Pakistan. Mercy Corps' paper health register was converted to a *Scan*-compatible format. Their health register had 39 data input fields, which expanded into 121 user data columns in the ODK database when factoring in metadata and image snippets for each field. Field workers used *Scan* to digitize patient records as they were filled in at the clinics, then validated the digitization with *Survey* at the point of capture. Additionally, a data reporting mechanism was developed on *Tables* for field workers to track overdue patient visits.

The large data sets generated by these facilities, coupled with unreliable Internet connectivity at the sites, provided scalability tests for the ODK 2's service architecture and synchronization capabilities. When Internet connectivity became available, usually at an Internet cafe, new data would be synchronized and used to generate administrative reports. The wide rows and extensive metadata for each patient caused the database size to grow, and with the spotty connectivity offered by Internet cafes, health workers struggled to upload the thousands of image files generated each day from the image segmentation of each question. Our responses to these challenges were discussed in Sections 4.3 and 4.5.

The custom data management application built on *Tables* provided health workers with a summary of patient data and reminders of upcoming appointments. This required disconnected historical queries of patients under the care of the health care worker using the device. Additionally, workers would often delay their validation of transcribed records until after a visit was complete, sometimes validating an entire day's worth of visits in a single batch. In this case, it was essential that workers had disconnected access to only their own patients to prevent them from mistakenly altering records of other patients. ODK 2's user and group permissions discussed in Section 4.3.3 were applied to limit the patient records available to the field worker to only the data they collected.

At the conclusion of the pilot, Mercy Corps interviewed its field workers about their experience and performed timing comparisons between workflows. The interviews discussed by Ali [2] revealed that workers were particularly enthused about the reporting on the device, with one stating "*follow up visit report has revolutionized our work. Now, we will never miss out any patient.*" Data collection was slowed by the need to validate transcribed data, but the transfer process was reduced from days via courier to minutes via connected synchronization [2]. The pilot showed that ODK 2 is robust enough to handle large datasets in disconnected environments and that organizations unable to fully adopt direct-to-digital workflows can benefit from field digitization.

### *5.2.2 Sync-Endpoint Web User Interface*

An example of ODK 2's modularity and extensibility was demonstrated by a non-profit organization called Benetech, whose goal is to empower communities with software for social good. Benetech partnered with Fundación Paraguaya (FP) to build a data collection platform to measure a family's poverty level. FP is a non-governmental organization focused on microfinance and entrepreneurship programs. FP developed a series of poverty indicators that assess to green-, yellow-, or red-status that were used to create the 'Poverty Spotlight' program. Benetech wanted to leverage open-source technologies to build a mobile application for the program and decided to use ODK 2 frameworks. Benetech made multiple extensions

to *Survey* to improve their ability to create visual surveys geared towards beneficiaries that have limited literacy, who were their primary end-users.

Originally, *Aggregate* was the default server for both ODK 1 and ODK 2. For Benetech's use case, they wanted a slimmed-down server that contained only the ODK 2 functionalities needed for their use case to avoid the complexities of a server with all the functionalities needed to support all of ODK 1 and ODK 2 use cases. Benetech took the subset of *Aggregate's* code that handled the bidirectional synchronization to create their own server named 'Hamster' (because hamsters are small). While Benetech was creating 'Hamster,' we had already forked *Aggregate* to change the underlying structure to abandon support for Google App Engine (GAE). We decided to remove GAE to upgrade the database abstraction layer to handle more complex queries with SQL. Since an abstraction layer can only provide the functionality that is provided by all the options that are abstracted behind the interface, it was necessary to remove GAE since it did not support SQL. GAE support was replaced with Microsoft SQL Server support since it allowed complex queries with SQL. This GAE-free server was named *Sync-Endpoint* because it focused on ODK 2's bi-directional synchronization and database design.

Initially, *Sync-Endpoint* kept Aggregate's web user interface. Benetech also designed a new user interface for their 'Hamster' server. However, Benetech designed 'Hamster' so that the web user interface was a separate web service from the synchronization service. Benetech's ability to implement a new user interface by merely relying on the ODK 2's REST interface abstraction without the need to access the database or other internal information directly exemplifies the ODK project's design principles 'Modularity' and 'Interoperability' discussed in Section 2.1.

Having an independent organization be able to utilize an existing public interface to obtain needed functionality demonstrates the effectiveness of ODK 2's abstractions for creating a modular and extensible system. Since Benetech's user interface followed ODK design principles, we decided to adopt their new user interface as part of the ODK 2 framework. As part of the redesign/replacement of Aggregate, ODK 2's default server was designed to be

deployed as a Docker [45] stack containing several Docker services to increase modularity. Using a Docker enables ODK 2 to easily package many open-source software platforms into a single deployable server. The multiple Dockers services running the various open-source software tools that comprise the functionality needed for a server facilitates a modular design that simplifies server deployment and maintenance. Adding the *Sync-Endpoint-Web-UI* as separate service included in a Docker stack improves ODK 2's modularity and demonstrates that designing for modularity with well documented public abstractions can encourage contributions.

### *5.2.3 Labor Market Panel Surveys*

An economics professor from St. Catherine University and her research team decided to use ODK 2 for Labor Market Panel Surveys (LMPS) because of ODK 2's ability to render complex workflows. The St. Catherine team conducts LMPS in low-resource contexts to research issues in development economics that primarily center on labor, education, health, and inequality in the Middle East and North Africa. Their projects focus on refugees, demographics, labor market dynamics, and human capital accumulation. The objective of an LMPS is to facilitate a better understanding of labor market dynamics and outcomes. LMPS can have approximately 2000-5000 questions depending on the goals of the researchers. Digital questionnaires are used to collect high-quality data because computers make it easy to enable question skip-logic that can accurately minimize the number of questions interviewees need to answer and to make sure the required questions are answered. Previous attempts to digitize LMPS led to problems such as not being able to connect individuals and households, poorly designed skip patterns, and missing responses to certain questions [7]. Researchers, in the *deployment architect* role, have experienced previous difficulties using other questionnaire software when attempting to express proper logic for question skip patterns. Instead of encoding all necessary logic for the workflow in their questionnaire, they simply included text indicators to the surveyor/*end-user* to interpret which questions to ask. Depending on the *end-user* to appropriately choose the relevant questions led to gaps in the data collection [7].

*Survey's* sub-form concept simplifies the design of LMPS questionnaires for households, as ODK 2 can automatically link questionnaires for individuals to their household to maintain data integrity. Additionally, since *Survey* exposes JavaScript functionality, the research team was able to express any logic they needed by having a full programming language available. This is in contrast to ODK 1's XForm specification and to the limited skip logic available in JavaRosa's custom relevance notation. Researchers with experience using statistics software were able to learn enough about HTML and JavaScript from online resources to encode LMPS *workflows* into ODK 2. The St. Catherine team did not receive direct support from the ODK research team beyond the public resources available to anyone wanting to use ODK 2. The St. Catherine team worked with the Egyptian government to conduct an LMPS using ODK 2 in Egypt in 2018 that collected approximately 60,000 responses [47]. The St. Catherine team intends to continue to use ODK 2 for future LMPS that are planned for Sudan in 2020 and Tunisia in 2020-21.

Using ODK 2 for various LMPS provides evidence of ODK 2's flexible frameworks' ability to digitize custom workflows for different use cases. The fact that an economics research team comprised of non-programmers was able to utilize ODK 2 abstractions without direct assistance from the ODK 2 development team provides evidence of ODK 2 ability to empower *deployment architects* to create custom mobile data applications. In addition to the St. Catherine's team successfully customizing ODK 2 to their workflow, they also created documentation and educational materials to teach others how to use and customize ODK 2. After successfully conducting ODK 2 training sessions, the St. Catherine team shared their training materials with the ODK community. After becoming an active member of the ODK community, the economics professor joined the technical steering committee and led a documentation refresh. The ability of a non-computer scientist to be able to self learn about how to customize ODK 2 frameworks, create educational material, and answer technical questions on the forum provides further evidence of ODK 2's success in creating multiple abstractions for different technical skill levels.

### *5.3 Humanitarian Disaster Response*

The International Federation of Red Cross and Red Crescent (IFRC) is the world's largest humanitarian organization with millions of volunteers in 190+ member National Societies. The diversity of use cases, languages, resources, and business requirements of the National Societies demonstrates the need for a flexible, customizable system. Since ODK 2 is designed to be configurable through its framework abstractions, it is a natural fit for the Red Cross movement as it enables reuse of core technology and training materials while empowering National Societies to customize to their unique circumstances. The IFRC's use of ODK 1 was limited to unidirectional data flows that do not provide the *end-user* (humanitarian field worker) with necessary data on their mobile devices for distributions or other decision-making tasks. This limitation created an issue for worker efficiency in many IRFC deployments (e.g., disaster response, migration crisis), as workers had to use two different sources of information to perform tasks. It is currently common for field workers to use both (1) a mobile device with ODK 1 for data entry and (2) either a laptop or paper to obtain additional information about a beneficiary and what they are entitled to receive.

The IFRC partnered with our research team to build a disaster response, disaster recovery, and crisis management system called "RC$^2$ Relief". RC$^2$ Relief is built upon ODK 2 frameworks, with pre-configured user interfaces designed for humanitarian relief *workflows*. These pre-configured interfaces implement predefined business logic and link beneficiary selection criteria with digitally collected profile data to reduce the amount of ODK 2 configuration needed when customizing *workflows* during a humanitarian response. RC$^2$ Relief contains several reference user interfaces for different disaster response scenarios with the various worker interfaces still accessing the same data structures. Since ODK 2 is a flexible rendering framework, *workflows* can be easily swapped in and out in times of crisis to change a volunteer's *workflow*. Having a reusable framework enables *deployment architects* to easily switch their field workers' *workflows* from a disaster response distribution to a health intervention. By helping *deployment architects* link predefined selection criteria with digitally

Figure 5.1: The photo shows one relief workers using a Android phone with ODK 1 and a other relief workers working side by side using paper to provide information and functionality that is not available with ODK 1. *Photo courtesy of the International Federation of Red Cross and Red Crescent Societies*

collected profile data, it enables humanitarian organizations to better target those in need. RC$^2$ Relief improves the ability of humanitarian organizations to better understand needs, distribute relief items, provide cash assistance, and conduct a variety of other humanitarian activities.

When designing RC$^2$ Relief, the IFRC specified four types of "users" based on already established job tasks. Since these four roles had different responsibilities, the data permission filters described in Section 4.3.3 provided vital functionality needed to satisfy the IFRC requirements. The four user roles for RC$^2$ Relief are:

- **Field Worker** – He/she uses the RC$^2$ Relief mobile app to perform *workflows* of basic field activities with beneficiaries such as collecting data, recording deliveries of items/cash, conducting follow-up data collections, and querying the database to answer beneficiary questions.

- **Field Supervisor** – He/she performs similar duties to the Field Worker but has the added responsibility for coordination of activities in the field, including filling the Field Report. He/she has extra privileges to be able to instruct the RC$^2$ relief mobile app to override the RC$^2$ Distribution Planning module decision on who should receive benefits.
- **Relief Coordinator** – He/she is responsible for the overall relief operation, including determining the criteria for beneficiaries to receive items or cash. The Relief Coordinator uses the RC$^2$ Distribution Planning module to pragmatically determine which beneficiaries are entitled to which benefits as well as create *workflows* for field workers to perform.
- **Information Management (IM) Officer** – He/she is responsible for the technical tasks, including server configuration and the setup of RC$^2$. He/she may need to provide technical support to the Relief Coordinator and assist with designing ODK 2 XLSX forms.

These established relief roles are consistent with the multi-perspective design discussed in Section 1.1 by splitting the roles of the *end-user* and the *deployment architect* in two, to create four roles. In the case of RC$^2$ Relief, both the Field Worker and Field Supervisor are *end-users* with different responsibilities. The data permission filters enabled the *end-user* to be programmatically split into different roles by providing additional privileged functionality to certain uses. Similarly, both the Relief Coordinator and IM Officer are *deployment architects* with different responsibilities and possibly different skill levels. When creating RC$^2$ Relief, the roles of *programmers* and *ODK framework developers* were performed by the ODK 2 research team. It is important to note that the four roles described in the multi-perspective design can be further divided up between multiple people or fulfilled by the same person. However, to make sure ODK 2 is usable by less-technical individuals, it is essential to recognize the different roles when creating abstraction layers to make sure appropriate interfaces for different skill levels are provided. Additionally, ODK 2's flexible permission scheme allows organizations to customize *workflows* to match the job responsibilities of different

users based on the requirements of their use cases.

The IFRC conducted an initial field feasibility study using ODK 2 to distribute disaster-aid cash at two locations in Jamaica involving 93 participants. The success of this feasibility study led to the RC$^2$ Relief tool being piloted in Syrian refugee camps in Greece (humanitarian assistance to the Syrian migration crisis). After piloting in Greece, the IFRC published literature that states: "*RC2 Relief may provide a time savings of up to 90% in the collection of information. This means more time is available to focus on delivery of better quality, more efficient, and personalized support to the communities that require it* [117]." The RC$^2$ Relief tool enabled field workers to utilize mobile devices to both register beneficiaries and subsequently deliver cash and other relief supplies. The RC$^2$ Relief tool improves the ability of humanitarian organizations to understand beneficiary needs, distribute relief items, provide cash assistance, aid in rebuilding programs, and conduct a variety of other humanitarian activities.

RC$^2$ Relief was released in January 2020 and has been used to respond to the Venezuelan Regional Migration Crisis and for Hurricane Dorian rebuilding efforts in the Bahamas. The Red Cross movement's field operations were curtailed in 2020 because of the COVID-19 virus global pandemic. To respond to the challenges of COVID-19, humanitarian *workflows* needed to adapt to new health requirements. RC$^2$ Relief's flexibility allowed the Red Cross movement to change *workflows* without requiring coding changes to RC$^2$ Relief, thus demonstrating ODK 2's flexibility for changing *workflows*. For example, in response to the Venezuelan regional migration crisis, cash distributions to Venezuelan migrants were successfully performed in Peru and Ecuador under COVID-19 restrictions with remote support. Thus, RC$^2$ Relief flexibility enabled National Societies to run a full relief cycle (assessment, registration, planning, distribution, and monitoring) with a limited presence on the field, minimizing contacts between communities and potential risks of virus transmission.

Based on the success of RC$^2$ Relief, the IFRC decided to expand the RC$^2$ project to handle non-relief *workflows*. The RC$^2$ Health project aims to develop an agile information management system capable of supporting both outpatient and community health campaigns

information management needs as well as intra-hospital management needs. The overall $RC^2$ Health concept is to create a modular and flexible tool that National Societies can adapt to the specific *workflows* of the country and health response.

### *5.4 Vaccine Cold-Chain*

National immunization programs are an effective public health intervention that is essential for controlling diseases. Central to immunization programs is the logistics system that distributes vaccines from global manufacturers to the point of use [153]. An essential component of vaccine programs is the vaccine cold chain equipment (cold rooms, refrigerators, freezers, cold boxes) used to store the vaccines. It is essential to have adequate cold storage space to ensure that vaccines are kept in an appropriate temperature range from arrival in the country until they are delivered to patients in order to maintain their potency. Vaccine refrigerators are specifically designed to keep temperatures in a specific range (2C to 8C), be robust to power failures, and to use different fuel sources based on the resources available in the local context.

Unfortunately, many existing cold chain equipment inventories were conducted with paper-based systems that contain large amounts of inaccurate or out-of-date information. Replacing paper-based inventory systems with a mobile Cold Chain Inventory application will improve the speed and reliability of the inventory update process. We partnered with PATH, WHO, and Gavi to create a 'vaccine cold chain inventory application' using ODK 2 frameworks. Immunization has been a global priority since 1974, when the World Health Organization (WHO) announced the Expanded Program on Immunization. WHO plays a global role in setting policies for immunization, supporting the introduction of new vaccines, and providing technical assistance. 'Gavi, the Vaccine Alliance' [53] is primarily a funder of vaccines, providing vaccines for free or at low cost to eligible countries. The goal of our partnership with WHO and Gavi is to create a mobile information management application that could be used by multiple countries to monitor and maintain a high-quality vaccine cold chain with sufficient working vaccine storage space. Similar to RC2 Relief, the design of

the mobile information management system needed to have flexible *workflows* and *dataflows* to adjust to the different country deployment specifics. Cold chain equipment is generally managed centrally by a country, with each country having a custom hierarchy of vaccine storage facilities. The hierarchy usually has a national vaccine warehouse that supplies regional storage facilities, which then distribute vaccines to hospitals, health centers, and clinics. The cold chain inventory application supports a set of cold chain management tasks that were determined by extensive discussions with country and global stakeholders [18]. The four categories of tasks are:

- **Inventory of Cold Chain Equipment -** Update the cold chain inventory by supporting collection of information about the health facilities as well as a list of the cold chain equipment present at the facility. Information collected about the health facilities generally includes location, type of facility, the role of a facility in the immunization system, the size of the population served by the facility, and infrastructure information such as the availability of electricity and other fuels.
- **Surveillance -** Regular tracking of refrigerator performance data such as the number of times the internal temperature goes out of an acceptable range to determine maintenance needs. Tracking equipment performance is also essential for studying the performance of new types of vaccine refrigerators such as solar direct drive refrigerators. Information about which models of refrigerators have the lowest rate of failures in specific field contexts could be easily determined if equipment data was available in a format that allowed for direct comparison.
- **Equipment Management -** Track repairs of equipment and allow the reporting of equipment needing repair. Countries need to plan for retirement and replacement of older equipment. Accurate knowledge of the vaccine cold chain allows identification of sites with inadequate storage and improved allocation of new resources. It also makes it possible to plan for the introduction of new vaccines.
- **Reporting -** Regular reporting of the status of cold chain equipment could improve vaccine delivery by providing up-to-date information on which vaccine refrigerators

are working and where resources should be directed to improve cold chain equipment. Having standardized reports will provide overview information on the status of the cold chain to inform country decision-makers as well as allow global organizations to compile reports for the global level.

A mobile information system that supports a remote workforce making real-time updates on location will significantly improve the inventory update process. Remote field workers can access up-to-date cold chain data with their mobile device while visiting sites that are disconnected from the Internet. These field workers use *Survey* and *Tables* to navigate *workflows* for updating facility and refrigerator information. In the cold chain application rendered by ODK 2 frameworks, the user navigates to health facilities via a geographic hierarchy or via a map-based interface if health facility coordinates are available. From the health facility pages, the user can access the lists of available refrigerators to update the information. Additional information is available to the user about the refrigerators, including maintenance records and model information. Being able to synchronize information with other field workers when Internet connectivity is available helps to decrease any duplication of work between field workers because a synchronized mobile device will have current information from all other workers' updates. As discussed in Section 4.3.2, to minimize data updates that conflict, data updates are processed as row-based changes to keep changes small. In the cold chain application, conflicts should be rare, as the technicians are working in geographically distinct areas and likely will not update the same rows of the tables within days.

By creating a mobile vaccine cold chain application with customizable *workflows* and *dataflows*, it enables organizations to leverage reusable code to make it easier for their use by 1) shrinking the number of options that have to be configured for the system to be used 2) minimizing the vocabulary being used to the domain space so the system features are more intuitive to users, and 3) having a set of predefined roles (e.g., immunization technicians, supervisors, ministry of health officials) that enable customized interfaces which display the appropriate information to the role. Similar to RC2 Relief, the cold chain application

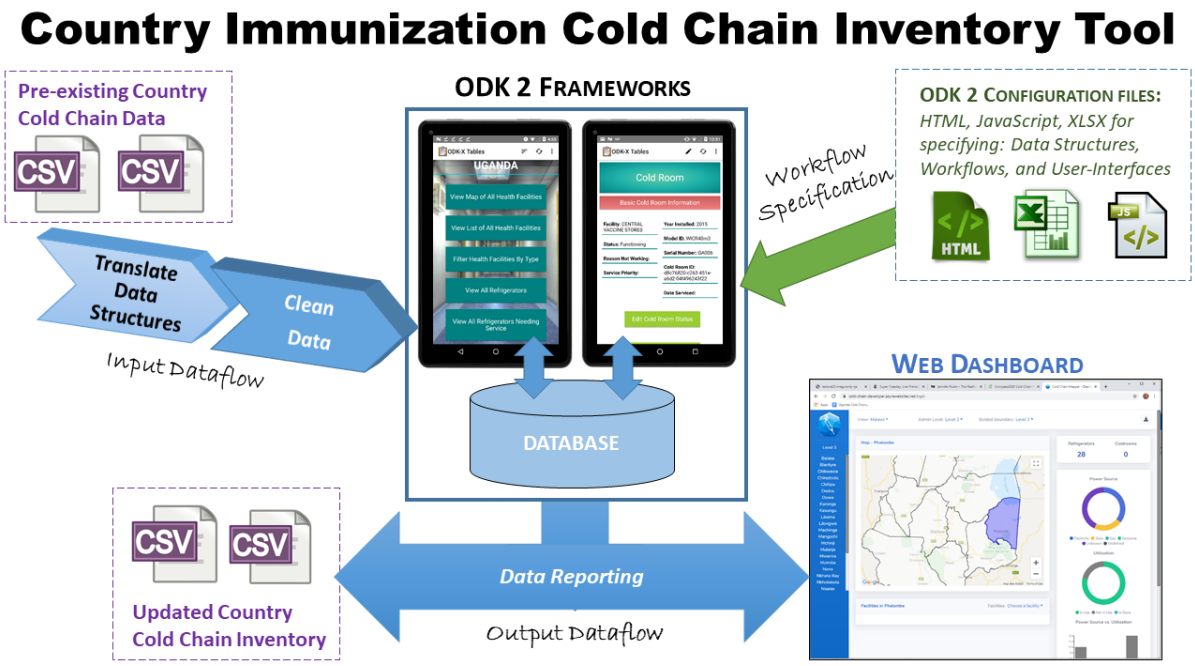



Figure 5.2: An architecture diagram of the immunization cold chain application. The *workflows* for various roles are specified by ODK 2 configuration files. The blue arrows shows the *dataflow* through the various pieces of software needed for import, update, and analysis [18].

leverages ODK 2 table-level and row-level permissions to create additional roles for *end-users* with different responsibilities and permissions. The *end-user* role is split into two roles; one for the cold chain technicians and one for their supervisors using permissions. The difference between the permission levels is that some operations (such as adding or deleting a health facility) are only available to supervisors or administrators. The cold chain application's *deployment architects* are the system administrators at the country level. Additional segmentation of *end-user* permissions is possible based on geographic regions (e.g., districts, states) since *end-users* are often assigned to teams that work in specific regions. ODK 2's group permission model provides countries the additional flexibility of basing permission on geographic regions, in order to limit data access.

To make a complete cold chain information management solution, our research team built additional software components to help with a country's *dataflow*. The entire vaccine cold

chain equipment information system includes three components 1) a data import tool to assist countries in importing their preexisting cold chain data; 2) a mobile application for field workers to update facility information (built with ODK 2 frameworks); 3) a web dashboard for visualization and summary data. The overall cold chain information architecture *dataflow* is shown in Figure 5.2. Unlike the ODK 2 frameworks that try to remain domain-independent, the web dashboard and visualization tools are designed as domain-dependent software that only targets vaccine cold chain use cases. We plan to extend the visualization and reporting system to support immunization campaigns with dashboards for cold chain capacity planning and tools for tracking and managing distributions of vaccines and reporting immunizations.

At the time of this dissertation publication, we have multiple cold chain pilot deployment projects underway. The WHO is currently piloting the ODK 2 cold chain equipment inventory application in the Democratic Republic of the Congo, Pakistan, and Bangladesh to monitor the performance of new equipment. Gavi is supporting a pilot project in Uganda to create a country-wide cold chain equipment inventory. Initially, Gavi supported a small 14 district cold chain inventory equipment update pilot. During this initial pilot, cold chain technicians performed facility information updates, recorded temperature monitoring results, and tracked refrigerator repairs. The initial three month pilot (February 2020 until April 2020) was successful and is being scaled up to a country-wide deployment in 2021. Results from the initial pilot showed that over 80% of the health care facilities' information was updated and over 80% of the cold chain equipment information was updated. All initial pilot participants who were interviewed reported that ODK 2 was either 'very easy' or 'moderately easy' to use.

# Chapter 6

# CONCLUSION

My dissertation focuses on building and evaluating software frameworks to empower users to construct mobile information services in low-resource contexts. The ODK research project seeks to identify 'appropriate' technologies for resource-constrained environments and identify barriers that prevent solutions that exist in high-resource contexts from being similarly adopted in these settings. Unfortunately, many technologies are designed for resource-rich environments where power and Internet connectivity are generally available. However, billions of people live in resource-poor contexts without Internet connectivity [151]. To address this "digital divide," research needs to be done about how to design technology to operate effectively in areas with limited infrastructure. System designers need to rethink some assumptions about network connectivity, power availability, user literacy, and device availability when designing systems for ubiquitous use anywhere in the world. The ODK project has become a catalytic innovation with over 2.6 million people in hundreds of countries using it to solve problems in diverse subject domains. Both the ODK 1 and ODK 2 tool suites enable organizations to create customized data collection and management applications.

As a co-founder of the ODK project, I had a lead role in the creation of the first tool suite. ODK 1 is a mobile data collection framework that has been adopted by a wide range of organizations (Chapter 2). A particular emphasis of my research was the evaluation of the ODK 1 tool suite and the creation of the ODK 2 tool suite. ODK 1 was designed to replace and enhance paper-based data collection *workflows*, so it focused on collecting data in a unidirectional *dataflow* (similar to paper) without functionality to synchronize data back to mobile clients for users to review and update. I gathered feedback from numerous organizations to evaluate the limitations of mobile data collection and management frameworks

(including ODK 1). Organizations reported missing functionality to support use cases where previously-collected data is often reviewed and updated (Section 3.1). Many organizations wanted a bidirectional *dataflow* to enable users to update previously collected data, provide field workers with necessary data to make decisions, and support more complex workflows based on previously collected data. This feedback led us to create a second ODK tool suite that focused on bidirectional "data management" *dataflows* instead of unidirectional "data collection" *dataflows* [20]. The ODK 2 tool suite was created to address the limitations uncovered through the widespread adoption of ODK 1. ODK 2 is runtime-adaptable to various application domains through the use of automation, configuration files, and runtime programming languages. ODK 2 frameworks have particular flexibility with regards to data input mechanisms, data models, workflows, and visual appearances (Chapters 3 & 4).

To generalize ODK frameworks design, we identified multiple actors in the application space (e.g., *programmer, deployment architect, end-user*) that interface with the frameworks using abstractions of differing complexity (Section 1.1). We also identified software system design characteristics that are needed to achieve both a global scale and be adaptable to local contexts (Section 1.2.1). The ODK 2 tool suite builds these design characteristics to provide organizations with configurable mobile frameworks designed to operate in resource-constrained environments. My dissertation research focused on designing, building, and evaluating multiple mobile software frameworks that provide abstractions that enable users (of varying skill levels) to configure and operate data collection and management applications in resource-constrained environments. Designing mobile applications for low-resource environments necessitates new abstractions that target *deployment architects*, non-developers who adapt the software to a deployment context. To provide the flexibility that organizations need to adapt mobile software frameworks to the local context requires the customizability of both the *workflow* and the *dataflow*. Data is integral to mobile application design and deployment and has both *inherent data characteristics* and *contextual data characteristics* that determine and restrict how data can be transmitted and stored.

I led the development of five modular ODK 2 frameworks - *Tables, Survey, Services,*

*Sensors, and Submit. Tables* is a data viewing framework that focuses on providing functionality to display, interact, curate, and update the entire data set. Users can explore the data through a variety of built-in views or create custom views using JavaScript/HTML. Reusable templates for viewing and organizing data through the framework's interface make it easier for organizations to design custom views to display and summarize data (Section 4.1). *Survey* is a runtime customizable question-rendering and constraint-verification framework that provides interactive, non-linear navigation capability and question widgets defined in JavaScript/HTML to enable runtime customization (Section 4.2). *Services* is a common data services framework that abstracts common functionality to create a service-oriented architecture. *Services* provides functionalities such as a web-server, data synchronization, and a single shared centralized database (Section 4.3). The *Sensors* framework simplifies both application and driver development with user-level abstractions that separate responsibilities between the user application, sensor framework, and device driver (Section 4.4). Finally, we investigated and built a prototype of the *Submit* framework to enable organizations to adapt to challenged network conditions (Section 4.5).

Implementing ODK 2 frameworks as a service-oriented architecture yielded benefits in the areas of upgradability, security enforcement, management of system state, and future portability, without incurring a significant performance cost. ODK 2 provides a second, complementary, open-source tool suite that is targeted at *deployment architects* and is easy to use and easy to scale. ODK 2 improves user experience through interactive, non-linear navigation to enable organizations to customize user interface complexity for differing levels of user expertise. ODK 2 expands the variety of use cases that the ODK project can support and helps to achieve ODK's overall goal of "magnifying human resources" through technology by providing open-source toolkits that are easy to try, easy to use, easy to modify, and easy to scale.

Building a 'real' system that has been used by organizations working in different subject domains shows the general applicability of the ODK 2 mobile frameworks. By piloting ODK 2 with organizations and incorporating their needs and feedback into the development process,

the frameworks were made robust to a variety of challenges met in the field, including extreme mobile networking conditions such as low bandwidth, high latencies, and long periods of disconnected activity. ODK 2 provides a generalizable application framework that allows users with varying technical skill levels to create customizable mobile data management solutions that address global problems in resource-constrained environments.

For mobile devices to be a useful and effective tool for solving global problems, researchers need to not only expand the capabilities of mobile technology but also expand how mobile technology can be adapted to varying environments with irregular constraints. This dissertation discusses lessons learned from creating application frameworks that make mobile devices more useful in contexts where high-income countries' engineering assumptions do not necessarily match the infrastructure realities in resource-constrained communities. Global development organizations need a system that minimizes the user's need to "know how to program" but still enables users to map their workflows, gather data, and digitize their decision making process into a flexible information management system. To empower a variety of users, ODK tries to minimize the computer skills needed to build mobile data applications. My research sought to understand what are the "usable" abstractions that enable users of various skill levels to control "runtime" adaptable systems through configuration files and non-compiled languages. Primarily I sought to understand how to create modular software frameworks that are composable by non-programmers, deployable by resource-constrained organizations, usable by minimally-trained users, and robust to intermittent power and networking outages.

## *6.1 Design Principles*

Despite many technological advances, technology is often unusable in many deployment contexts because of design assumptions relating to available resources and infrastructure. For example, the assumption of constant connectivity to access cloud data has resulted in a 'digital divide,' as humanitarian organizations that operate in areas of limited connectivity (where over half the world's population reside[151]) do not have the same technical require-

ments as users in resource-rich environments. The unavailability of suitable software tools can be a limiting factor to the scale and complexity of the services provided to beneficiaries in certain locations. A challenge for many global development organizations is finding 'appropriate' technology designed to operate effectively in remote locations where vulnerable populations require assistance.

Effective use of technology and digital data can help organizations increase efficiency and provide better service to beneficiaries. Organizations working in resource-constrained communities often rely on mobile devices to amplify their field workers' productivity. Mobile devices can provide digital information at the point of delivery, thus enabling field workers and field managers to make better-informed decisions. Adoption of digital technologies can lead to revolutions in efficiency and cost-effectiveness; however, for technology to have a profound impact, it needs to be designed for broad reuse. Three key design decisions were made to enable ODK's reuse and to allow organizations to scale technology solutions without the involvement of the ODK research team. The three key design decisions were 1) creating a modular design, 2) following commercial technology trends, and 3) creating a generic platform.

### *6.1.1 Modular Design*

A common developer debate is whether to use a modular design or a singular monolithic design. The book *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* [116] discusses modular design and the debate between top-down versus bottom-up design. In the case of global development, modular design and monolithic design both have strengths and weaknesses. A modular design enables organizations to have the flexibility and adaptability to compose a customized information system by using different components that best fit the local context. Modular design simplifies parallel development and enables new functionality to be released in pieces over time instead of having to deliver everything at once. However, users and developers often find custom monolithic solutions to be more straightforward, as the core software can be modified to do 'exactly what the user

wants,' and developers can optimize performance and workflows. Unfortunately, customized monolithic solutions are often domain-specific tools that can be inflexible to customization for reuse in another similar domain. This inability to customize to different contexts encourages the proverbial 're-inventing the wheel.' Additionally, custom solutions often have custom data storage and schema, keeping data siloed in the monolithic solution.

To encourage the creation of multiple software modules, the ODK project uses open standards and a permissive, open-source software license to enable anyone to improve the system. By being open and modular, the barriers to contribution are lower as developers can reuse most of ODK's functionality, thus saving time and resources. People can start with the basic functionality that works already and can customize modules for specific use cases. Since ODK is a modular open-source project, many commercial companies have taken the base modules to add features and create additional value for customers. Enabling companies to create value and new products based on core ODK modules helped the ODK project to scale. Giving others (e.g., commercial companies) an incentive to improve ODK to meet the needs of organizations helps ODK scale since our small research team is unable to meet the requests of all organizations.

The ODK 2 frameworks aim to satisfy an organization's specific usage scenario through the composition of narrowly focused tools rather than a single monolithic app. An app that tries to provide abstractions for every usage scenario can be overwhelming, so ODK 2 frameworks focus on specific functionalities. The frameworks are designed to smoothly transition between one another to give the feel of a unified application. This enables organizations to compose a solution that only uses the tools that are 'appropriate' for their deployments. Our goal was to make sure the frameworks could work either independently or together, to enable organizations to make a simple system with a couple of frameworks or more complex systems with multiple frameworks. This approach is in contrast to developing a highly customized single-purpose application that is only useful for limited use cases.

### *6.1.2 Follow Technology Trends*

ODK was designed to target commercially available mobile devices and cloud platforms to simplify an organization's ability to scale. By using 'appropriate' cutting-edge technology, it enables organizations to leverage the global capital that is spent on research and development to drive innovation and keep costs low by manufacturing at scale. If a small research team produced custom technology that was not supported by the global marketplace, there would be little momentum to drive the price lower and create 'appropriate' versions of the technology for different contexts. In 2008-2009, the ODK project ignored the advice of many global development experts that kept focusing on low-cost solutions and chose Android devices as an 'appropriate' cutting-edge technology for low-resource communities. Simply focusing on the price of technology at a particular point in time to create a low-cost solution misses how technology evolves and can become a low-cost solution. The ODK project chose Android because it was a new open platform that was encouraging manufacturers to produce a variety of form factors with different price points. This meant that companies would be applying their resources to solve problems for international development organizations, such as localizing the technology to different users, creating multiple form factors for different use cases, and creating different price points appropriate to the contexts. After adopting ODK, organizations found that, while the mobile device was more expensive, the amount the organization saved through increased worker efficiency and decreased worker training made ODK a lower-cost solution [62]. Technology solutions should follow technology trends to avoid early obsolescence and leverage global market forces. Early obsolescence can lead to "pilot-itius" as organizations will conduct new pilots to verify that the technology replacing the 'obsolete and unavailable technology' will meet organizational requirements. The lowest price point technology may not have the lowest overall cost, as the technology needs to be readily available 'on-the-shelf' for 5 to 10 years for a solution to be able to be scaled.

### *6.1.3 Generic Platform*

Technology innovations designed as a reusable generalized platform can act as a innovation catalyst, leading to rapid adoption and growth because of the platform's reusability and flexibility. Software systems can either be designed to be a platform for generic use or be a highly customized solution for a specific use case. As discussed in Chapter 2, an exemplar of a generalized platform with broad adoption is Microsoft Office. Office has a generalized design that is not specifically targeted at a single country, nor does it target customers in a single domain. Office is a useful software tool for organizations working in different subject domains in many countries.

The ODK project enables organizations to reuse technology across traditional boundaries of subject domain, connectivity, borders, language, etc. By providing abstractions that organizations can use to customize the platform to their needs, the ODK project encourages reusability between multiple development organizations and promotes cost-savings. Having an adaptable open-source software platform for data management can significantly reduce costs through 1) reuse and sharing of a generalizable platform, 2) shared software development cost, 3) better information on which to make evidence-based decisions at the global office, regional office, field manager, and field worker levels, and 4) decreased training and acquisition costs.

## *6.2 Remaining Data Challenges*

The promise of machine learning has opened many possibilities of computer-assisted decision making and has led to research on how to process "big data." However, these techniques could also be applied to smaller sets of data to help humanitarian organizations improve their impact. The future of global development data analysis could be seen as applying "big data" tools to small datasets to make it easier for practitioners to better leverage technology. Data needs to be collected, updated, and processed to be able to be analyzed by machine learning algorithms or by humans simply using summary data to make informed

decisions. The ODK project thus far has focused on collecting and updating data, which often requires domain-independent functionality that can be adapted to a domain through custom *workflows*. The ODK 1 tool suite focused on data collection, while the ODK 2 tool suite focused on data management, which includes both collecting and updating data. The ODK project has historically only provided basic data analysis tools because data analysis is often domain-dependent making it more difficult to create a generic platform.

### *6.2.1 Data Cleaning and Analysis*

The continual improvement of technology and the digitization of vast amounts of survey and sensor data are promising more reliable, real-time information for global development and humanitarian organizations. Visualization and decision support can help an organization become more nimble in their projects and make smarter decisions closer to real-time (within hours and days of data collection, rather than months or years). The ability to use this data, however, is hampered by the inability to process this disparate data in a timely manner. To overcome this challenge, the global development and humanitarian community need a complete end-to-end data pipeline to not just capture and aggregate raw data, but also process it into meaningful results and provide actionable outputs through compelling visualizations and robust analysis. The requirements in Section 3.1.1 outline the essential elements for building an end-to-end pipeline for data cleaning and analysis.

While Pervaiz [109] outlined a taxonomy of data cleaning in low-resource regions and created algorithms for data cleaning transliterate names, more work needs to be done. The research field of 'Data Science' is making progress on automating data cleaning and analysis; however, additional research is required in order to make sure the algorithms work for different languages and contexts. Beyond data cleaning, there are additional concerns about data analysis for humanitarian organizations that work with vulnerable populations. Organizations often collect personal information, like pregnancy status or human rights violations, to determine the types of services a beneficiary should receive. However, sending this personal data to a cloud storage location presents legal and ethical issues. These data

sensitivity issues can create significant barriers to organizations using commercially available options. Some commercial products have options to host data locally; however, leveraging these options often increases costs.

Another issue with data analysis is data sharing, as organizations and researchers often wish to maintain ownership and privacy over their datasets, complicating their efforts to share information across the various silos that exist in the global development community. Data is stored in a shared repeatable format and can be transformed into other formats to enable interoperability with existing big data visualization technology. A new data provenance tracking algorithm is needed to ensure that original datasets remain accessible to enable a truly collaborative community of researchers and development actors to share datasets, learn from others' interventions, and build upon prior successes to increase the efficacy of future interventions.

### *6.2.2 Conflict Resolution and Data Merging*

A problem that has not been fully solved by ODK 2 is how to handle frequent changes to the data model, as experienced by the World Mosquito Program (WMP) that was discussed in Sections 3.5.5 & 5.1.3. Initial ODK 2 design discussions pitted a document-based data model against a row-based model. The document-based model allows for easier updates to the data model, but extracting results and generating reports after multiple data models have been used becomes difficult and often requires hiring a programmer to write code to merge the data and perform data cleaning. The row-based model employed by ODK 2 requires organizations to establish a schema before deploying, front-loading the costs. However, it is more difficult to alter the table schema, but reporting is simple because everything is already in a linked tabular format. Ultimately, the row-based model was chosen because organizations have more existing expertise with tabular data formats, and making a flexible schema requires less programming knowledge than dealing with data merging and cleaning issues. However, a scenario like WMP's, where frequent schema changes happen, confirmed that if organizations have programming resources to handle the data merging and cleaning,

then a document-based model can make for easier iterative design.

Organizations often request features to make it easy to add or delete questions to a questionnaire. While technically not difficult, it means that the datastore needs to change to accommodate more information. The challenge comes when the system needs to interpret how to merge the data. Since *workflows* digitized with ODK tools are often dependent on the data for logic decisions, question skip-logic, or worker instructions, having incomplete data causes *workflow* issues. When asking organizations about what behavior should happen when data is missing or information needs to be merged. The answers from organizations on how to merge the data were not consistent, as it depended on the deployment context and the organization's goals. We failed to find any commonality on how to perform data merging that could be generalized. Without a general approach, it is difficult to make a reusable tool because each case requires some customization to handle the multiple conditions, which may only be able to be expressed in a programming language. The problem of data merging is akin to resolving conflicts in a database as the correct way to resolve the conflict requires understanding contextual information and the implications of the assumptions and decisions.

### *6.3 Final Remarks*

Two-thirds of the world's population has access to mobile phones; however, technical limitations prevent billions of users from adopting some of the transformative capabilities of these mobile devices [16, 151]. The concept of 'applied computer science research' is needed to extend the reach of technology to include the billions of people in the world that currently have limited access to it. Research is needed to understand "what are the proper abstractions and frameworks for mobile devices that are appropriate for different contexts?" as well as "what abstractions are appropriate for a broad range of technical skills and familiarity?" Adaptable frameworks that create abstractions that target application-level users are needed to empower *deployment architects* to customize their application to match an organization's requirements. For mobile tools to be successful in resource-constrained environments, they

should be composable by non-programmers, deployable by resource-constrained organizations, usable by minimally trained users, and robust to intermittent power and networking outages.

Interdisciplinary and intradisciplinary research is needed to build and evaluate computing technologies that assist humanity in addressing global challenges such as healthcare, humanitarian relief, and environmental sustainability. The ODK project is at the intersection of many research areas, including mobile computing, computing for social good, human-computer interaction, edge computing, computer-supported cooperative work, human-centered design, economics, medicine, public health, and public policy. Specifically, my research uses consumer devices, sensors, and reusable software services to create computer systems that integrate seamlessly into a user's environment. This research vision is similar to Mark Weiser's vision: *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it"* [149]. Technology should be available to everyone in the world so anyone can leverage the efficiency gains. However, for technology to 'weave themselves into everyday life,' work is needed so that 'system' assumptions match constraints experienced by the world's population. To identify and solve these types of issues needs both 'systems' and 'human-computer interaction' research, as the topics are inter-related and more value should be placed on bridging the gap to solve real technology issues instead of leaving it to the 'other' field to handle.

Much of ODK's success came from the decision to have researchers partner with organizations and work side-by-side in the deployment contexts. This interaction gave the developers real-time, culturally appropriate, in-situ feedback that would not have emerged if the researcher had not been in the field. Conceptually akin to testbeds (e.g., PlanetLab [29]), by working with organizations to build real systems that are deployed in regions with challenging constraints, we were able to test both ODK 1 and ODK 2 beyond the lab environment. Deploying systems in various contexts helped reveal system design assumptions that limited functionality because of incorrect expectations about deployment contexts. Often small assumptions made in an abstraction layer can make a technology inappropriate

for an organization's use. Having a *real* system operated by a humanitarian organization in a low-resource context enables the identification of problematic design assumptions and cross-abstraction layer issues.

Many of the technical issues in low-resource contexts do not present a fundamentally new class of technology problems. However, the technology issues demonstrated that the engineering decisions and design assumptions made when creating a system can have a significant impact on the usability and adaptability of a system to other contexts.The common assumptions used for high-resource environments used by designers and developers (e.g., connectivity and power availability) can be inappropriate for resource-constrained environments. For example, the availability of power can vary significantly between different contexts, as availability of power could vary from an always-on grid connection, to sporadic grid connections, to having to pay someone to use a generator or car battery to charge a mobile device. Also, depending on your location in the world, mobile devices often move from disconnected environments to connected environments with dramatically different bandwidths and latency. The ODK tool suites are designed to operate in extremely diverse environments. The lessons learned making ODK adaptable to different contexts can be applied generally to system design to make mobile system platforms more resilient to adverse and varying conditions.

# BIBLIOGRAPHY

[1] Adobe. PhoneGap. http://phonegap.com/. Accessed September 2020.

[2] Syed Ali, Rachel Powers, Jeffrey Beorse, Arif Noor, Farah Naureen, Naveed Anjum, Muhammad Ishaq, Javariya Aamir, and Richard Anderson. ODK Scan: Digitizing Data Collection and Impacting Data Management Processes in Pakistan's Tuberculosis Control Program. *Future Internet*, 8(4):51, 2016.

[3] Ruth Anderson, Beth Kolko, Laura Schlenke, Waylon Brunette, Alexis Hope, Rob Nathan, Wayne Gerard, Jacqueline Keh, and Michael Kawooya. The Midwife's Assistant: Designing Integrated Learning Tools to Scaffold Ultrasound Practice. In *Proceedings of the Fifth International Conference on Information and Communication Technologies and Development*, ICTD '12, pages 200–210, New York, NY, USA, 2012. ACM.

[4] Apache. Cordova. https://cordova.apache.org/. Accessed September 2020.

[5] Apache. CouchDB. http://couchdb.apache.org/. Accessed September 2020.

[6] Arduino. http://www.arduino.cc/. Accessed September 2020.

[7] Ragui Assaad, Samir Ghazouani, Caroline Krafft, and Dominique J Rolando. Introducing the Tunisia Labor Market Panel Survey 2014. *IZA Journal of Labor & Development*, 5(1):1–21, 2016.

[8] Backbone.js. https://backbonejs.org/. Accessed September 2020.

[9] Paramvir Bahl, Atul Adya, Jitendra Padhye, and Alec Walman. Reconsidering Wireless Systems with Multiple Radios. *SIGCOMM Computing Communication Review*, 34(5):39–46, October 2004.

[10] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting Mobile 3G Using WiFi. In *Proc of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 209–222, 2010.

[11] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI Replication. In *NSDI*, volume 6, pages 5–5, 2006.

[12] Abhishek Bhardwaj, Pandarasamy Arjunan, Amarjeet Singh, Vinayak Naik, and Pushpendra Singh. MELOS: A Low-Cost and Low-Energy Generic Sensing Attachment for Mobile Phones. In *Proceedings of the 5th ACM Workshop on Networked Systems for Developing Regions*, NSDR '11, page 27–32, New York, NY, USA, 2011. Association for Computing Machinery.

[13] Jørn Braa, Ole Hanseth, Arthur Heywood, Woinshet Mohammed, and Vincent Shaw. Developing Health Information Systems in Developing Countries: The Flexible Standards Strategy. *MIS Q.*, 31(2):381–402, June 2007.

[14] Jørn Braa and Calle Hedberg. The Struggle for District-Based Health Information Systems in South Africa. *The Information Society*, 18(2):113–127, 2002.

[15] Jørn Braa and Sundeep Sahay. The DHIS2 Open Source Software Platform: Evolution Over Time and Space. In Leo Anthony, G. Celi, Hamish S. F. Fraser, Vipan Nikore, Juan Sebastian Osorio, and Kenneth Paik, editors, *Global Health Informatics. Principles of eHealth and mHealth to Improve Quality of Care*. The MIT Press, 04 2017.

[16] E. Brewer, M. Demmer, B. Du, M. Ho, M. Kam, S. Nedevschi, J. Pal, R. Patra, S. Surana, and K. Fall. The Case for Technology in Developing Regions. *IEEE Computer*, 38(6):25–38, 2005.

[17] E. Brewer, M. Demmer, M. Ho, R.J. Honicky, J. Pal, M. Plauche, and S. Surana. The Challenges of Technology Research for Developing Regions. *IEEE Pervasive Computing*, 5(2):15–23, 2006.

[18] Waylon Brunette, Clarice Larson, Shourya Jain, Aeron Langford, Yin Yin Low, Andrew Siew, and Richard Anderson. Global goods software for the immunization cold chain. In *Proceedings of the 3rd ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS '20, page 208–218, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Waylon Brunette, Rita Sodt, Rohit Chaudhri, Mayank Goel, Michael Falcone, Jaylen Van Orden, and Gaetano Borriello. Open Data Kit Sensors: A Sensor Integration Framework for Android at the Application-level. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 351–364, New York, NY, USA, 2012. ACM.

[20] Waylon Brunette, Samuel Sudar, Mitchell Sundt, Clarice Larson, Jeffrey Beorse, and Richard Anderson. Open Data Kit 2.0: A Services-Based Application Framework for Disconnected Data Management. In *Proceedings of the 15th Annual International*

*Conference on Mobile Systems, Applications, and Services*, MobiSys 2017, pages 440–452, New York, NY, USA, 2017. ACM.

[21] Waylon Brunette, Samuel Sudar, Nicholas Worden, Dylan Price, Richard Anderson, and Gaetano Borriello. ODK Tables: Building Easily Customizable Information Applications on Android Devices. In *Proceedings of the 3rd ACM Symposium on Computing for Development*, ACM DEV '13, pages 12:1–12:10, New York, NY, USA, 2013. ACM.

[22] Waylon Brunette, Mitchell Sundt, Nicola Dell, Rohit Chaudhri, Nathan Breit, and Gaetano Borriello. Open Data Kit 2.0: Expanding and Refining Information Services for Developing Regions. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Waylon Brunette, Morgan Vigil, Fahad Pervaiz, Shahar Levari, Gaetano Borriello, and Richard Anderson. Optimizing Mobile Application Communication for Challenged Network Environments. In *Proceedings of the 2015 Annual Symposium on Computing for Development*, DEV '15, pages 167–175. ACM, 2015.

[24] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano. Device-to-Device Communications with Wi-Fi Direct: Overview and Experimentation. *Wireless Communications, IEEE*, 20(3):96–104, June 2013.

[25] R. Chaudhri, R. Sodt, K. Lieberg, J. Chilton, G. Borriello, Y. J. Masuda, and J. Cook. Sensors and Smartphones: Tracking Water Collection in Rural Ethiopia. *IEEE Pervasive Computing*, 11(3):15–24, 2012.

[26] Rohit Chaudhri, Eleanor O'Rourke, Shawn McGuire, Gaetano Borriello, and Richard Anderson. FoneAstra: Enabling Remote Monitoring of Vaccine Cold-Chains Using Commodity Mobile Phones. In *Proceedings of the First ACM Symposium on Computing for Development*, ACM DEV '10, New York, NY, USA, 2010. Association for Computing Machinery.

[27] Rohit Chaudhri, Darivanh Vlachos, Jabili Kaza, Joy Palludan, Nathan Bilbao, Troy Martin, Gaetano Borriello, Beth Kolko, and Kiersten Israel-Ballard. A System for Safe Flash-Heat Pasteurization of Human Breast Milk. In *Proceedings of the 5th ACM Workshop on Networked Systems for Developing Regions*, NSDR '11, page 9–14, New York, NY, USA, 2011. Association for Computing Machinery.

[28] Kuang Chen, Akshay Kannan, Yoriyasu Yano, Joseph M. Hellerstein, and Tapan S. Parikh. Shreddr: Pipelined Paper Digitization for Low-resource Organizations. In

*Proceedings of the 2Nd ACM Symposium on Computing for Development*, ACM DEV '12, pages 3:1–3:10, New York, NY, USA, 2012. ACM.

[29] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[30] Byung-Gon Chun, Carlo Curino, Russell Sears, Alexander Shraer, Samuel Madden, and Raghu Ramakrishnan. Mobius: unified messaging and data serving for mobile apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 141–154, 2307650, 2012. ACM.

[31] Camille Cobb, Samuel Sudar, Nicholas Reiter, Richard Anderson, Franziska Roesner, and Tadayoshi Kohno. Computer security for data collection technologies. *Development Engineering*, 3:1 – 11, 2018.

[32] Andrew Cross, Nakull Gupta, Brandon Liu, Vineet Nair, Abhishek Kumar, Reena Kuttan, Priyanka Ivatury, Amy Chen, Kshama Lakshman, Rashmi Rodrigues, George D'Souza, Deepti Chittamuru, Raghuram Rao, Kiran Rade, Bhavin Vadera, Daksha Shah, Vinod Choudhary, Vineet Chadha, Amar Shah, Sameer Kumta, Puneet Dewan, Bruce Thomas, and William Thies. 99DOTS: A Low-cost Approach to Monitoring and Improving Medication Adherence. In *Proceedings of the Tenth International Conference on Information and Communication Technologies and Development*, ICTD '19, pages 15:1–15:12, New York, NY, USA, 2019. ACM.

[33] D3: Data-Driven Documents. http://digitalprinciples.org/. Accessed September 2020.

[34] Tathagata Das, Prashanth Mohan, Venkata N. Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. PRISM: Platform for Remote Sensing Using Smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, page 63–76, New York, NY, USA, 2010. Association for Computing Machinery.

[35] Lara B. Deek, Kevin C. Almeroth, Mike P. Wittie, and Khaled A. Harras. Exploiting Parallel Networks Using Dynamic Channel Scheduling. In *Proc of the 4th Annual International Conference on Wireless Internet*, WICON '08, pages 1–9, ICST, Brussels, Belgium, Belgium, 2008.

[36] Nicola Dell, Nathan Breit, Timóteo Chaluco, Jessica Crawford, and Gaetano Borriello. Digitizing paper forms with mobile imaging technologies. In *Proceedings of the 2nd*

*ACM Symposium on Computing for Development*, ACM DEV '12, pages 1–10. ACM, 2012.

[37] Nicola Dell, Trevor Perrier, Neha Kumar, Mitchell Lee, Rachel Powers, and Gaetano Borriello. Paper-Digital Workflows in Global Development Organizations. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '15, pages 1659–1669, 2675145, 2015. ACM.

[38] Shuo Deng, Anirudh Sivaraman, and Hari Balakrishnan. All Your Network Are Belong to Us: A Transport Framework for Mobile Network Selection. In *Proc of the 15th Workshop on Mobile Computing Systems & Applications*, HotMobile '14, 2014.

[39] B. DeRenzi, C. Sims, J. Jackson, G. Borriello, and N. Lesh. A Framework for Case-Based Community Health Information Systems. In *2011 IEEE Global Humanitarian Technology Conference*, pages 377–382, Oct 2011.

[40] Brian DeRenzi, Gaetano Borriello, Jonathan Jackson, Vikram S. Kumar, Tapan S. Parikh, Pushwaz Virk, and Neal Lesh. Mobile Phone Tools for Field-Based Health care Workers in Low-Income Countries. *Mount Sinai Journal of Medicine: A Journal of Translational and Personalized Medicine*, 78(3):406–418, 2011.

[41] Brian DeRenzi, Nicola Dell, Jeremy Wacksman, Scott Lee, and Neal Lesh. Supporting Community Health Workers in India Through Voice- and Web-Based Feedback. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 2770–2781, New York, NY, USA, 2017. ACM.

[42] Brian DeRenzi, Neal Lesh, Tapan Parikh, Clayton Sims, Werner Maokla, Mwajuma Chemba, Yuna Hamisi, David S hellenberg, Marc Mitchell, and Gaetano Borriello. e-IMCI: improving pediatric health care in low-income countries. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 753–762, 1357174, 2008. ACM.

[43] Brian DeRenzi, Jeremy Wacksman, Nicola Dell, Scott Lee, Neal Lesh, Gaetano Borriello, and Andrew Ellner. Closing the Feedback Loop: A 12-month Evaluation of ASTA, a Self-Tracking Application for ASHAs. In *Proceedings of the Eighth International Conference on Information and Communication Technologies and Development*, ICTD '16, pages 22:1–22:10, New York, NY, USA, 2016. ACM.

[44] Digital Green. http://www.digitalgreen.org/. Accessed September 2020.

[45] Docker. http://www.docker.com. Accessed September 2020.

[46] DropBox. https://www.dropbox.com/. Accessed September 2020.

[47] Egypt Labor Market Panel Survey, ELMPS 2018. http://www.erfdataportal.com/index.php/catalog/157, November 2019. Accessed September 2020.

[48] Faulkner et al. HTML 5. https://www.w3.org/TR/html52/, December 2017. Accessed September 2020.

[49] Jon Froehlich, Mike Y. Chen, Sunny Consolvo, Beverly Harrison, and James A. Landay. MyExperience: A System for in Situ Tracing and Capturing of User Feedback on Mobile Phones. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 57–70, New York, NY, USA, 2007. ACM.

[50] FrontlineSMS. https://www.frontlinesms.com/. Accessed September 2020.

[51] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 168–178, New York, NY, USA, 2008. Association for Computing Machinery.

[52] R. Gandhi, R. Veeraraghavan, K. Toyama, and V. Ramprasad. Digital Green: Participatory video for agricultural extension. In *2007 International Conference on Information and Communication Technologies and Development*, pages 1–10. IEEE, 2007.

[53] Gavi, the Vaccine Alliance. https://www.gavi.org/. Accessed September 2020.

[54] Amy Sarah Ginsburg, Jaclyn Delarosa, Waylon Brunette, Shahar Levari, Mitch Sundt, Clarice Larson, Charlotte Tawiah Agyemang, Sam Newton, Gaetano Borriello, and Richard Anderson. mPneumonia: Development of an Innovative mHealth Application for Diagnosing and Treating Childhood Pneumonia and Other Childhood Illnesses in Low-Resource Settings. *PloS one*, 10(10), 2015.

[55] Amy Sarah Ginsburg, Charlotte Tawiah Agyemang, Gwen Ambler, Jaclyn Delarosa, Waylon Brunette, Shahar Levari, Clarice Larson, Mitch Sundt, Sam Newton, Gaetano Borriello, and Richard Anderson. mPneumonia, an Innovation for Diagnosing and Treating Childhood Pneumonia in Low-Resource Settings: A Feasibility, Usability and Acceptability Study in Ghana. *PLOS ONE*, 11(10), 2016.

[56] Google. Connect People Everywhere. http://www.google.com/loon/. Accessed January 2019.

[57] Google. Google Drive. https://www.google.com/drive/. Accessed September 2020.

[58] Network Working Group. Atom Syndication Format. http://www.ietf.org/rfc/rfc4287.txt, December 2005. Accessed September 2020.

[59] Bo Han, Pan Hui, V. S. A Kumar, M. V Marathe, Jianhua Shao, and A Srinivasan. Mobile Data Offloading Through Opportunistic Communications and Social Participation. *IEEE Transactions on Mobile Computing*, 11(5):821–834, May 2012.

[60] Handlebars. https://handlebarsjs.com/. Accessed September 2020.

[61] S. Hao, N. Agrawal, A. Aranya, and C. Ungureanu. Building a delay-tolerant cloud for mobile data. In *2013 IEEE 14th International Conference on Mobile Data Management*, volume 1, pages 293–300, 2013.

[62] Carl Hartung, Yaw Anokwa, Waylon Brunette, Adam Lerer, Clint Tseng, and Gaetano Borriello. Open Data Kit: Tools to Build Information Services for Developing Regions. In *Proceedings of the 4th ACM/IEEE International Conference on Information and Communication Technologies and Development*, ICTD '10, New York, NY, USA, 2010. Association for Computing Machinery.

[63] Richard Heeks. Information Systems and Developing Countries: Failure, Success, and Local Improvisations. *The Information Society*, 18(2):101–112, 2002.

[64] Richard Heeks. Health information systems: Failure, success and improvisation. *International journal of medical informatics*, 75:125–37, 03 2006.

[65] Richard Heeks. Avoiding eGov Failure: Design-Reality Gap Techniques. www.egov4dev.org/success/techniques/drg.shtml, 2008.

[66] Kurtis Heimerl, Kashif Ali, Joshua Blumenstock, Brian Gawalt, and Eric Brewer. Expanding Rural Cellular Networks with Virtual Coverage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 283–296, Lombard, IL, 2013. USENIX.

[67] John Hicks, Nithya Ramanathan, Donnie Kim, Mohamad Monibi, Joshua Selsky, Mark Hansen, and Deborah Estrin. AndWellness: An Open Mobile System for Activity and Experience Sampling. In *Wireless Health 2010*, WH '10, page 34–43, New York, NY, USA, 2010. Association for Computing Machinery.

[68] YoonSung Hong, Hilary K. Worden, and Gaetano Borriello. ODK Tables: Data Organization and Information Services on a Smartphone. In *Proceedings of the 5th ACM Workshop on Networked Systems for Developing Regions*, NSDR '11, page 33–38, New York, NY, USA, 2011. Association for Computing Machinery.

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

[70] Fei Huang, Sean Blaschke, and Henry Lucas. Beyond Pilotitis: taking digital health interventions to the national level in China and Uganda. *Globalization And Health*, 13(1), 2017.

[71] Intel. XDK. https://software.intel.com/intel-xdk, 2017. Accessed September 2020.

[72] Ionic. Ionic Framework. http://ionicframework.com/. Accessed September 2020.

[73] David L. Johnson, Elizabeth M. Belding, and Consider Mudenda. Kwaabana: File Sharing for Rural Networks. In *Proceedings of the 4th Annual Symposium on Computing for Development*, ACM DEV-4 '13, pages 4:1–4:10, New York, NY, USA, 2013. ACM.

[74] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating Hardware Device Failures in Software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 59–72, New York, NY, USA, 2009. Association for Computing Machinery.

[75] Asim Kadav and Michael M. Swift. Understanding Modern Device Drivers. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 87–98, New York, NY, USA, 2012. Association for Computing Machinery.

[76] Bonifaz Kaufmann and Leah Buechley. Amarino: A Toolkit for the Rapid Prototyping of Mobile Ubiquitous Computing. In *Proceedings of the 12th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '10, pages 291–298, New York, NY, USA, 2010. ACM.

[77] Andreas Kipf, Waylon Brunette, Jordan Kellerstrass, Matthew Podolsky, Javier Rosa, Mitchell Sundt, Daniel Wilson, Gaetano Borriello, Eric Brewer, and Evan Thomas. A proposed integrated data collection, analysis and sharing platform for impact evaluation. *Development Engineering*, 1:36 – 44, 2016.

[78] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1):3–25, February 1992.

[79] Klaus Krickeberg. Principles of Health Information Systems in Developing Countries. *Health Information Management Journal*, 36(3):8–20, 2007.

[80] Ye-Sheng Kuo, Sonal Verma, Thomas Schmid, and Prabal Dutta. Hijacking Power and Bandwidth from the Mobile Phone's Audio Interface. In *Proceedings of the First ACM Symposium on Computing for Development*, ACM DEV '10, pages 24:1–24:10, New York, NY, USA, 2010. ACM.

[81] Leaflet. https://leafletjs.org/. Accessed September 2020.

[82] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yue-Ting Shen, Kevin Elphinstone, and Gernot Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5):654–664, 2005.

[83] Laura Dan Li and Jay Chen. TroTro: Web Browsing and User Interfaces in Rural Ghana. In *Proceedings of the Sixth International Conference on Information and Communication Technologies and Development: Full Papers - Volume 1*, ICTD '13, pages 185–194, New York, NY, USA, 2013. ACM.

[84] Yong Li, Guolong Su, Pan Hui, Depeng Jin, Li Su, and Lieguang Zeng. Multiple Mobile Data Offloading Through Delay Tolerant Networks. In *Proc of the 6th ACM Workshop on Challenged Networks*, CHANTS '11, pages 43–48, 2011.

[85] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two Years of Experience with a (Mu)-Kernel Based OS. *SIGOPS Oper. Syst. Rev.*, 25(2):51–62, April 1991.

[86] Felix Xiaozhu Lin, Ahmad Rahmati, and Lin Zhong. Dandelion: A Framework for Transparently Programming Phone-Centered Wireless Body Sensor Applications for Health. In *Wireless Health 2010*, WH '10, page 74–83, New York, NY, USA, 2010. Association for Computing Machinery.

[87] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. Reflex: Using Low-power Processors in Smartphones Without Knowing Them. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 13–24, New York, NY, USA, 2012. ACM.

[88] Henry Lucas. Information and communications technology for future health systems in developing countries. *Social Science & Medicine*, 66(10):2122–2132, 2008.

[89] Burke W Mamlin, Paul G Biondich, Ben A Wolfe, Hamish Fraser, Darius Jazayeri, Christian Allen, Justin Miranda, and William M Tierney. Cooking up an open source EMR for developing countries: OpenMRS - a recipe for successful collaboration. *AMIA Annual Symposium proceedings*, pages 529–533, 2006.

[90] Marcelo Martins and Rodrigo Fonseca. Application Modes: A Narrow Interface for End-user Power Management in Mobile Devices. In *Proc of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, 2013.

[91] Mercy Corps. https://www.mercycorps.org/. Accessed September 2020.

[92] Microsoft. OneDrive. https://onedrive.live.com/. Accessed September 2020.

[93] Marc Mitchell, Maya Getchell, Melania Nkaka, Daniel Msellemu, Jan Van Esch, and Bethany Hedt-Gauthier. Perceived Improvement in Integrated Management of Childhood Illness Implementation through Use of Mobile Technology: Qualitative Evidence From a Pilot Study in Tanzania. *Journal of Health Communication*, 17(sup1):118–127, 2012.

[94] Marc Mitchell, Bethany L Hedt-Gauthier, Daniel Msellemu, Melania Nkaka, and Neal Lesh. Using electronic technology to improve clinical care–results from a before-after cluster trial to evaluate assessment and classification of sick children according to Integrated Management of Childhood Illness (IMCI) protocol in Tanzania. *BMC medical informatics and decision making*, 13(1):95, 2013.

[95] Marc Mitchell and Lena Kan. Digital Technology and the Future of Health Systems. *Health Systems & Reform*, 5(2):113–120, 2019.

[96] Marc Mitchell, Neal Lesh, Hilarie Cranmer, Hamish Fraser, Irina Haivas, and Kate Wolf. Improving care–improving access: the use of electronic decision support with AIDS patients in South Africa. *International Journal of Healthcare Technology and Management*, 10(3):156–168, 2009.

[97] Henry Mwanyika, David Lubinski, Richard J. Anderson, Kelley Chester, Mohamed Makame, Matt Steele, and Don de Savigny. Rational Systems Design for Health Information Systems in Low-Income Countries: An Enterprise Architecture Approach. *Journal of Enterprise Architecture*, 7(4), 2011.

[98] MyMedLab. http://www.mymedlab.com/, 2013. Accessed September 2020.

[99] S. Nirjon, A. Nicoara, Cheng-Hsin Hsu, J. Singh, and J. Stankovic. MultiNets: Policy Oriented Real-Time Switching of Wireless Interfaces on Mobile Devices. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 251–260, April 2012.

[100] Oracle Database Mobile Server. https://www.oracle.com/database/technologies/related/mobile-server.html. Accessed September 2020.

[101] World Health Organization. Integrated Management of Childhood Illness (IMCI). https://www.who.int/maternal_child_adolescent/topics/child/imci/en/l. Accessed September 2020.

[102] Joyojeet Pal, Anjuli Dasika, Ahmad Hasan, Jackie Wolf, Nick Reid, Vaishnav Kameswaran, Purva Yardi, Allyson Mackay, Abram Wagner, Bhramar Mukherjee, Sucheta Joshi, Sujay Santra, and Priyamvada Pandey. Changing Data Practices for Community Health Workers: Introducing Digital Data Collection in West Bengal, India. In *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development*, ICTD '17, pages 17:1–17:12, New York, NY, USA, 2017. ACM.

[103] Tapan S. Parikh, Paul Javid, Sasikumar K., Kaushik Ghosh, and Kentaro Toyama. Mobile phones and paper documents: evaluating a new approach for capturing microfinance data in rural India. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 551–560, 1124857, 2006. ACM.

[104] Tapan S. Parikh and Edward D. Lazowska. Designing an Architecture for Delivering Mobile Information Services to the Rural Developing World. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 791–800, New York, NY, USA, 2006. ACM.

[105] PATH. https://www.path.org/. Accessed September 2020.

[106] PATH. Digital Square Global Goods Guidebook. https://static1.squarespace.com/static/59bc3457ccc5c5890fe7cacd/t/5ced6f3c7817f7e261ddbc0a/1559064401781/Global-Goods-Guidebook_V1.pdf, 2019. Accessed September 2020.

[107] Rabin K Patra, Sergiu Nedevschi, Sonesh Surana, Anmol Sheth, Lakshminarayanan Subramanian, and Eric A Brewer. WiLDNet: Design and Implementation of High Performance WiFi Based Long Distance Networks. In *Proc of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 7–7, Berkeley, CA, USA, 2007.

[108] Seneca Perri-Moore, Thomas Routen, Amani Flexson Shao, Clotide Rambaud-Althaus, Ndeniria Swai, Judith Kahama-Maro, Valerie D'Acremont, Blaise Genton, and Marc Mitchell. Using an eIMCI-Derived Decision Support Protocol to Improve Provider–Caretaker Communication for Treatment of Children Under 5 in Tanzania. *Global Health Communication*, 1(1):41–47, 2015.

[109] Fahad Pervaiz. *Understanding challenges in the data pipeline for development data*. University of Washington, Seattle, 2019.

[110] Principles for Digital Development. http://digitalprinciples.org/. Accessed September 2020.

[111] World Mosquito Program. The World Mosquito Program. https://www.worldmosquitoprogram.org/. Accessed September 2020.

[112] QR Code. http://www.qrcode.com/en/about/version.html. Accessed September 2020.

[113] Chris Quintana, Brian J. Reiser, Elizabeth A. Davis, Joseph Krajcik, Eric Fretz, Ravit Golan Duncan, Eleni Kyza, Daniel Edelson, and Elliot Soloway. A Scaffolding Design Framework for Software to Support Science Inquiry. *Journal of the Learning Sciences*, 13(3):337–386, 2004.

[114] Divya Ramachandran, John Canny, Prabhu Dutta Das, and Edward Cutrell. Mobilizing Health Workers in Rural India. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1889–1898, New York, NY, USA, 2010. ACM.

[115] RapidSMS. https://www.rapidsms.org//. Accessed September 2020.

[116] Eric S Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, Incorporated, Sebastopol, 2001.

[117] RC2 Relief. https://media.ifrc.org/ifrc/wp-content/uploads/sites/5/2019/12/RCR-Leaflet-V3-20191206.pdf. Accessed September 2020.

[118] Matthew J Renzelmann and Michael M Swift. Decaf: Moving Device Drivers to a Modern Language. In *USENIX Annual Technical Conference*, 2009.

[119] IDC Research. Sartphone OS Market Share, 2019. https://www.idc.com/prodserv/smartphone-os-market-share.jsp, June 2020. Accessed September 2020.

[120] David W Richardson and Steven D Gribble. Maverick: Providing Web Applications with Safe and Flexible Access to Local Devices. In *WebApps*, 2011.

[121] Johan Saebo and Ola Titlestad. Evaluation of a bottom-up action research approach in a centralised setting: HISP in Cuba. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences,*, volume 37, page 11 pp., 02 2004.

[122] Safaricom. M-pesa. https://www.safaricom.co.ke/personal/m-pesa. Accessed September 2020.

[123] Sundeep Sahay and John Lewis. Strengthening Metis Around Routine Health Information Systems in Developing Countries. *Information Technologies & International Development*, 6(3), 2010.

[124] Sundeep Sahay, Eric Monteiro, and Margunn Aanestad. Toward a political perspective of integration in information systems research: The case of health information systems in India. *Information Technology for Development*, 15(2):83–94, 2009.

[125] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An Analysis of Internet Content Delivery Systems. *SIGOPS Oper. Syst. Rev.*, 36(SI):315–327, December 2002.

[126] James Scott, Jon Crowcroft, Pan Hui, and Christophe Diot. Haggle: A Networking Architecture Designed Around Mobile Users. In *WONS 2006: Third Annual Conference on Wireless On-demand Network Systems and Services*, pages 78–86, 2006.

[127] Rahul C. Shah, Sumit Roy, Sushant Jain, and Waylon Brunette. Data MULEs: Modeling and Analysis of a Three-tier Architecture for Sparse Sensor Networks. *Ad Hoc Networks*, 1(2–3):215 – 233, 2003.

[128] Saureen Shah and Apurva Joshi. COCO: A Web-based Data Tracking Architecture for Challenged Network Environments. In *Proc of the First ACM Symposium on Computing for Development*, ACM DEV '10, pages 1–7, 2010.

[129] Amani Flexson Shao, Clotilde Rambaud-Althaus, Ndeniria Swai, Judith Kahama-Maro, Blaise Genton, Valerie D'Acremont, and Constanze Pfeiffer. Can smartphones and tablets improve the management of childhood illness in Tanzania? A qualitative study from a primary health care worker's perspective. *BMC health services research*, 15(1):135, 2015.

[130] Jacob Sorber, Nilanjan Banerjee, Mark D. Corner, and Sami Rollins. Turducken: Hierarchical Power Management for Mobile Devices. In *Proceedings of the 3rd International*

*Conference on Mobile Systems, Applications, and Services*, MobiSys '05, page 261–274, New York, NY, USA, 2005. Association for Computing Machinery.

[131] SQLite. http:/sqlite.org. Accessed September 2020.

[132] R. W. Stevens and G. R. Wright. TCP/IP Illustrated: Vol. 2: The Implementation, 1995.

[133] Samuel R. Sudar and Richard Anderson. DUCES: A Framework for Characterizing and Simplifying Mobile Deployments in Low-Resource Settings. In *Proceedings of the 2015 Annual Symposium on Computing for Development*, DEV '15, pages 23–30, New York, NY, USA, 2015. ACM.

[134] Tablecast. http://tablecast.org/. Accessed September 2020.

[135] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[136] Kentaro Toyama. Technology as Amplifier in International Development. In *Proceedings of the 2011 IConference*, iConference '11, page 75–82, New York, NY, USA, 2011. Association for Computing Machinery.

[137] Heather Underwood, S. Revi Sterling, and John K. Bennett. The Design and Implementation of the PartoPen Maternal Health Monitoring System. In *Proceedings of the 3rd ACM Symposium on Computing for Development*, ACM DEV '13, pages 8:1–8:10, New York, NY, USA, 2013. ACM.

[138] International Telecommunications Union. The World in 2011: ICT Facts and Figures. https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2011-e.pdf, 2011. Accessed 2015.

[139] International Telecommunications Union. The World in 2013: ICT Facts and Figures. https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013-e.pdf, 2013. Accessed September 2020.

[140] Ushahidi. https://https://www.ushahidi.com/. Accessed September 2020.

[141] Sarah Van Wart, K. Joyce Tsai, and Tapan Parikh. Local Ground: A Paper-based Toolkit for Documenting Local Geo-spatial Knowledge. In *Proceedings of the First ACM Symposium on Computing for Development*, ACM DEV '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM.

[142] Vidado. https://vidado.ai/. Accessed September 2020.

[143] VillageReach. https://www.villagereach.org/. Accessed September 2020.

[144] VillageReach. The Framework for OpenLMIS. http://www.villagereach.org/wp-content/uploads/2012/03/02292012.Framework-for-OpenLMIS-Whitepaper.pdf, February 2012. Accessed September 2020.

[145] Nicolas Villar, James Scott, and Steve Hodges. Prototyping with Microsoft .Net Gadgeteer. In *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction*, TEI '11, page 377–380, New York, NY, USA, 2010. Association for Computing Machinery.

[146] Andrii Vozniuk, Adrian Holzer, Sten Govaerts, Jorge Mazuze, and Denis Gillet. Graspeo: A Social Media Platform for Knowledge Management in NGOs. In *Proceedings of the Seventh International Conference on Information and Communication Technologies and Development*, ICTD '15, pages 63:1–63:4, New York, NY, USA, 2015. ACM.

[147] Andrii Vozniuk, Adrian Holzer, Jorge Mazuze, and Denis Gillet. GraaspBox: Enabling Mobile Knowledge Delivery into Underconnected Environments. In *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development*, ICTD '17, pages 13:1–13:11, New York, NY, USA, 2017. ACM.

[148] Lu Wei-Chih, Matt Tierney, Jay Chen, Faiz Kazi, Alfredo Hubard, Jesus Garcia Pasquel, Lakshminarayanan Subramanian, and Bharat Rao. UjU: SMS-based Applications Made Easy. In *Proceedings of the First ACM Symposium on Computing for Development*, ACM DEV '10, pages 1–11. ACM, 2010.

[149] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–105, 1991.

[150] OpenRosa API WorkGroup. OpenRosa WorkGroup API Page: OpenRosa 1.0 APIs. https://bitbucket.org/javarosa/javarosa/wiki/OpenRosaAPI, December 2011. Accessed September 2020.

[151] World Bank Group. *World Development Report 2016: Digital Dividends*. International Bank for Reconstruction and Development (World Bank), 1818 H Street NW, Washington DC, 20433, 2016.

[152] Kok-Kiong Yap, Te-Yuan Huang, Masayoshi Kobayashi, Yiannis Yiakoumis, Nick McKeown, Sachin Katti, and Guru Parulkar. Making Use of All the Networks Around

Us: A Case Study in Android. In *Proc of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*, CellNet '12, pages 19–24, 2012.

[153] M. Zaffran, J. Vandelaer, D. Kristensen, B. Melgaard, P. Yadav, K. O. Antwi-Agyei, and H. Lasher. The imperative for stronger vaccine supply and logistics systems. *Vaccine*, 31:B73–B80, April 2013.

[154] Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. Improving Energy Efficiency of Location Sensing on Smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, page 315–330, New York, NY, USA, 2010. Association for Computing Machinery.