A Synchronization Protocol for ODK Tables and ODK Aggregate

Dylan Price

A senior thesis submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science
With Departmental Honors
Computer Science & Engineering
University of Washington
June 2012

Presentation of work given on June 1, 2012

Thesis and presentation approved by ________________________________

Date ______________________
1 Problem

Smartphones are becoming more and more prevalent as a key tool for ICTD work in the developing world. Tools such as Open Data Kit (ODK) Collect [1] have opened the doors to a wide array of possibilities.

With these increasingly powerful tools available, organizations want projects to have more sophisticated data collection capabilities that go beyond simply replacing paper forms with a convenient UI on the phone. Some of these capabilities include exploring and updating complex, relational data on the phone and sharing data across multiple phones.

In addition, tools that solve these problems must be easy to deploy and maintain, and flexible enough to work in very different environments.

The focus of this paper is a data synchronization system for the developing world which leverages existing ODK tools. It is a flexible, open source solution which will enable organizations in the developing world to give workers in the field the ability to share data between phones, as well as save the data to a server or the cloud. This in turn enables these organizations to pursue more sophisticated projects.

This system takes advantage of two ODK tools: ODK Tables [2] and ODK Aggregate [3]. ODK Tables is an Android [4] smartphone application that enables exploring and updating complex, relational data on the phone. Before this synchronization system was designed, however, there was no easy way to connect data in Tables with a backend server. ODK Aggregate was chosen as this backend. It is a web service that provides a data repository for forms as well as a way to visualize, export and publish this data. The synchronization protocol is built as an HTTP API on top of the existing Aggregate codebase, and takes advantage of its security and database agnostic storage layer. Aggregate is also container agnostic, meaning it can be deployed to the cloud or a private server.

In this paper, I will first relate the main requirements of the synchronization protocol, then I will discuss some existing technologies that meet some of these requirements but fail to meet all of them in a significant way, that is, fail significantly enough that development of a new technology was needed. After that I will explain the solution I have built and some of the important design choices that came up while building it. Finally, I will evaluate my solution against the original requirements to show how I have achieved the original goal and areas that still need work or could be improved.

2 Requirements

The requirements of this project evolved from a set of use cases for ICTD projects in the developing world. The general model of these use cases is an organization with many field workers who need to share a single dataset. The workers may need to view and update this dataset in the field, and the managers need to be able to restrict access to the dataset so that field workers can only see and edit the parts that are relevant to them. What follows is the requirements of the synchronization protocol.

Work with ODK Tables. ODK Tables is an application for Android that allows exploring and updating moderately complex, relational datasets. A brief overview of it is given later in the paper. We chose to build the synchronization protocol primarily for ODK
Tables because it was a familiar code base that we could easily adapt to our needs, and will hopefully become an emerging tool for ICTD work in the developing world.

**Back up data and metadata.** First and foremost, the synchronization protocol should allow data and metadata to be saved to and restored from a server.

**Share data across phones and keep it synchronized.** Besides simply backing up data, it should be possible for multiple phones to access and modify the same set of data, and the data should stay in sync.

**Robust to low network connectivity.** Users in the developing world often have limited Internet connectivity. In these situations, the synchronization protocol should not break, but should degrade gracefully.

**Extensible to other synchronization backends.** It should be possible to adapt the synchronization protocol on the phone to work with different backends.

**Authentication based on Google accounts.** The synchronization protocol should reuse the Google accounts infrastructure for authentication of users.

**Support a simple mechanism for groups of users.** Administrators should be able to create groups of users and set access controls for these groups.

**Provide moderately complex and fine-grained access control that allows both table and row-level permissions.** Administrators should be able to restrict access to tables and metadata. Besides restricting access on a table wide level, administrators should also be able to enforce a filtered view of a table for a certain user or group of users.

**Straightforward conflict resolution.** When concurrent modifications are made to the same piece of data (i.e. the same row of a table), there should be a simple and reliable mechanism for resolving the conflicting updates.

**Simple, easy to understand HTTP API.** The API which is exposed by the backend should be simple to understand and use. Other applications may wish to use it in the future.

**Maintain complete history of changes to a table.** When rows are created, updated, or deleted, the history of these changes should be saved and accessible to administrators.

**Easy to deploy and maintain.** Many organizations in the developing world do not have the ability to take on sophisticated IT responsibilities. The solution should be simple for less technical users to set up and keep running.

**Affordable solution for smaller organizations.** Small organizations can not afford to pay a lot of money on software or server cycles. The solution should keep costs to the end user to a minimum.

**Scalable to moderately sized datasets and deployments.** The synchronization protocol should be able to work smoothly for tables with thousands of rows and hundreds of phones.

### 3 Related Work

Before deciding on implementing a new system I looked at a number of existing technologies that could be adapted to serve a similar purpose. For one reason or another, none of them completely met the requirements. What follows is a discussion of each technology I looked at and the main drawbacks that prevented it from being used to implement the synchronization protocol.
3.1 Apache CouchDB

CouchDB [5] is a "database that completely embraces the web." As such, it has built in access control and synchronization features and is accessible through an HTTP API.

However, CouchDB is a json document-based database and does not cleanly map to the relational model of data in ODK Tables. The document paradigm means that each database is simply a collection of documents which are independent and schema free. Since Tables is based on the concept of a spreadsheet with typed columns, it is hard to map this to a representation in CouchDB that is efficient and not overly complicated. Since the relational model is so well understood, it makes little sense to take on this additional and unnecessary complexity.

Additionally, CouchDB requires an organization to maintain their own servers and software stack. For many organizations in the developing world this kind of infrastructure is out of reach.

3.2 Google Fusion Tables

Google Fusion Tables [6] is a service offered as part of Google Docs which provides access to simple cloud-based tables and a sql-like HTTP API. It is simple and free to use.

The big disadvantage of Fusion Tables for the synchronization protocol is the inability to insert processing logic into the data path between a phone running ODK Tables and the backend database. This limits the possibilities to the basic features that the Fusion Tables API provides. For instance, Fusion Tables does not provide any mechanisms for controlling concurrent access to the table, does not expose its access control model through an API, and does not provide permissions on a finer level than a whole table.

3.3 Tablecast

Tablecast [7] is an xml protocol built on top of the Atom syndication format [8]. It is designed specifically for dealing with concurrent modifications to tables of data.

However, because it is based on the Atom specification, Tablecast relies on a publish/subscribe architecture where each participant acts as both a client and a server. Modifications to a table are published as a feed and multiple feeds from multiple publishers can be merged together to see the latest state of a table. This is a bad architecture for the requirements of the synchronization protocol because phones, with their limited battery life and network connectivity, are unreliable as providers of feeds. Furthermore, the architecture would require phones to talk directly with each other, which is error prone and cumbersome for the same reasons.

3.4 Amazon S3

Amazon’s S3 [9] is a cloud-based service for storing data. It comes with sophisticated access control and versioning features that would make implementation of many of the requirements of the synchronization protocol straightforward.
But, as with CouchDB, S3 does not use the relational model for storing data, rather, it uses the notion of buckets and objects. This is very close to the CouchDB’s databases and documents, except that objects may be any data whatsoever and are not required to be json documents. This means there are no querying capabilities and no systems for maintaining a table with a schema. The burden falls on the application installed on the phone, making its responsibilities and interactions with the backend unnecessarily complex.

Secondly, S3 does not offer a free tier. Thus for small organizations and projects who can’t afford to pay, a synchronization protocol based on S3 would preclude them from being able to take advantage of it.

3.5 Oracle Database Mobile Server 11g

Oracle’s Database Mobile Server [10] fulfills almost all of the stated requirements. It uses a relational model to synchronize data across mobile phones and to a backend server. It has sophisticated access control, handles concurrent updates and conflict resolution, and allows inserting arbitrary processing logic into the data path.

While it meets all of these needs, Oracle’s Database Mobile Server is ultimately proprietary and very expensive. Each organization must make its own private contract with Oracle so it is not a scalable solution in terms of reaching many different organizations. It is also out of reach financially for many projects in the developing world.

4 Description of Solution

This section describes the synchronization protocol and discusses a few of the design challenges faced while building it. It starts out with some general background and context on ODK Tables and ODK Aggregate, the two systems used to implement the synchronization protocol. After that is a brief overview of what the protocol is and how it works. Finally, three design challenges faced while building the protocol are discussed. These are the access control and table filtering mechanism, the synchronization state on the phone, and the conflict resolution strategy.

4.1 ODK Tables

ODK Tables is an application for the Android operating system that is designed for exploring and updating relational datasets on a smartphone or tablet. Tables maintains a list of tables the user can view and modify. Each table is like a simple database table, with typed columns and rows of data. These tables may be viewed in many different ways, including as raw tabular data, lists of rows, and graphs. Additionally, tables may be linked to each other to form a relationship between them, much like a JOIN in a sql database. More information on ODK Tables can be found at [2].

Given this general design, there are multiple elements that need to be synchronized with a backend. Specifically, there are both data, i.e. the rows of a table, and metadata, i.e. the name, schema, and other properties that define it.
4.2 ODK Aggregate

ODK Aggregate is a web service built to store, visualize, and publish form-based data. For example, an organization might use Aggregate to store filled out surveys, then publish them to a Google Spreadsheet [11].

At a lower level, Aggregate also provides a database and container agnostic Java web application. That is, it can be run in Google App Engine [12] or Apache Tomcat [13], and on top of the App Engine datastore, MySQL [14], or PostgreSQL [15]. This flexibility is provided as distinct layers below the main functionality of Aggregate and it is on top of these lower layers that the synchronization protocol is built. For more information on ODK Aggregate, see [3].

4.3 General Overview

The synchronization protocol provides a mechanism to keep a table stored in ODK Aggregate in sync across multiple phones running ODK Tables. The basic architecture is shown in Figure 1. It consists of an HTTP API exposed by Aggregate that one or more phones running Tables contact to synchronize a table. The HTTP API is based on some of the principles of a REST architecture, therefore it defines a set of resources that can be manipulated using common HTTP verbs such as GET, PUT, and DELETE. All operations are either strongly idempotent, meaning repeating the same invocation multiple times produces the exact same side effects, or weakly idempotent, meaning any repetitions of an invocation after the first one will return an error. This greatly simplifies the job of clients to the API because they do not have to worry about losing the results of an API call, they may always safely repeat the request. Authentication is handled using Google accounts and the OAuth2 [16] protocol.

Concurrent access to the rows of a table are handled with a form of optimistic concurrency control, where one user’s changes are accepted and all others rejected. When a phone makes a request to update a row in a table in Aggregate, Aggregate locks the entire table and executes the update, which includes updating the version of the row, called the ‘row etag’. Any concurrent updates must wait for the lock, and when they get it and try to update that same row they will find that their row version no longer matches the version of the row in Aggregate. Their update will consequently be rejected and the user of the phone must resolve the conflict between their changes and the row on the server before trying to update again.
The top-level resource of the HTTP API is a table. Under this are the subresources rows, diff, columns, properties, and acl. Figures 2 and 3 detail the operations available on these resources and Table 1 shows the types used to represent them.
<table>
<thead>
<tr>
<th>Name</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TableEntry</td>
<td><code>&lt;tableEntry&gt;</code>&lt;tableId&gt;1&lt;/tableId&gt;&lt;tableName&gt;people&lt;/tableName&gt;&lt;dataEtag&gt;3&lt;/dataEtag&gt;&lt;propertiesEtag&gt;4&lt;/propertiesEtag&gt;`</td>
</tr>
<tr>
<td>TableDefinition</td>
<td><code>&lt;tableDefinition&gt;</code>&lt;name&gt;people&lt;/name&gt;&lt;column name=&quot;name&quot; type=&quot;STRING&quot;/&gt;&lt;column name=&quot;age&quot; type=&quot;INTEGER&quot;/&gt;&lt;column name=&quot;weight&quot; type=&quot;DECIMAL&quot;/&gt;&lt;metadata&gt;ODK Tables metadata here&lt;/metadata&gt;`</td>
</tr>
<tr>
<td>Column</td>
<td><code>&lt;column name=&quot;name&quot; type=&quot;STRING&quot;/&gt;</code></td>
</tr>
<tr>
<td>Row</td>
<td><code>&lt;row&gt;</code>&lt;id&gt;1&lt;/id&gt;&lt;etag&gt;2&lt;/etag&gt;&lt;deleted&gt;false&lt;/deleted&gt;&lt;entry column=&quot;weight&quot;&gt;175.0&lt;/entry&gt;&lt;entry column=&quot;age&quot;&gt;23&lt;/entry&gt;&lt;entry column=&quot;name&quot;&gt;dylan&lt;/entry&gt;`</td>
</tr>
<tr>
<td>TableProperties</td>
<td><code>&lt;tableProperties&gt;</code>&lt;etag&gt;4&lt;/etag&gt;&lt;name&gt;people&lt;/name&gt;&lt;metadata&gt;ODK Tables metadata here&lt;/metadata&gt;`</td>
</tr>
<tr>
<td>TableAcl</td>
<td><code>&lt;tableAcl&gt;</code>&lt;scope&gt;&lt;type&gt;USER&lt;/type&gt;&lt;value&gt;<a href="mailto:user@gmail.com">user@gmail.com</a>&lt;/value&gt;&lt;/scope&gt;&lt;role&gt;READER&lt;/role&gt;`</td>
</tr>
</tbody>
</table>

Table 1: Types used in the HTTP API of the synchronization protocol.
/tables
GET Get a list of all TableEntry the user has permission to read.

/tables/{tableId}
GET Get the TableEntry specified by tableId.
PUT Create a new table with the tableId. Request body should contain a TableDefinition with a name, a string of metadata, and a set of columns. Returns the TableEntry for the new table (this is the only asymmetric API call, that is, what it returns is different than what is PUT).
DELETE Delete the table (and all associated Columns, Rows, TableAcls, and TableProperties).

/tables/{tableId}/columns
GET Get a list of Columns for the table.

/tables/{tableId}/properties
GET Retrieve the TableProperties for the table.
PUT Set the TableProperties for the table.

/tables/{tableId}/rows
GET Get a list of all current Rows for the table.

/tables/{tableId}/rows/{rowId}
GET Get the Row specified by rowId.
PUT Create or update a Row.
DELETE Delete a Row.

/tables/{tableId}/diff?data_etag={data_etag}
GET Compute changes since the table’s data entity tag was at the given value.

Figure 2: Synchronization API resources and operations, part one.
/tables/{tableId}/acl
GET Get a list of all TableAcl for the table.

/tables/{tableId}/acl/default
GET Get the default scoped TableAcl.
PUT Set the default scoped TableAcl.
DELETE Delete the default scoped TableAcl.

/tables/{tableId}/acl/user
GET Get all TableAcl scoped to specific users.

/tables/{tableId}/acl/user/{userId}
GET Get the TableAcl scoped to the user with userId.
PUT Set the TableAcl scoped to the user with userId.
DELETE Delete the TableAcl scoped to the user with userId.

/tables/{tableId}/acl/group
GET Get all TableAcl scoped to specific groups.

/tables/{tableId}/acl/group
GET Get all TableAcl scoped to specific groups.

/tables/{tableId}/acl/group/{groupId}
GET Get the TableAcl scoped to the group with groupId.
PUT Set the TableAcl scoped to the group with groupId.
DELETE Delete the TableAcl scoped to the group with groupId.

Figure 3: Synchronization API resources and operations, part two.
To see the protocol in action, consider the simple example outlined in Figure 4. In this example a phone is synchronizing an existing table with a server running ODK Aggregate. The server has three changes the phone doesn’t have, and the phone has both a row to insert and a row to update on the server.

First, the phone retrieves the TableEntry from the server. This tells it the latest entity tags for both data and properties of the table. As seen in Table 1, the data and properties entity tags are simply integers which are incremented every time a change is made to the table data or properties, respectively. With the latest entity tags, the phone now knows whether data or metadata on the server has changed since the last synchronization. For this scenario, the phone starts at a data etag of two and a properties etag of two.

Since the server is at a data etag of five, the phone requests a diff representing all changes to the rows of the table since the table was at data etag two. The server responds with a list of the three rows that have changed. The phone then figures out if each row is a new row it has never seen before or an update to a row it already has, and inserts or updates the row accordingly.

Now the phone is up to date with the state of the server, so it is able to push its changes up to the server. It has one row to update and one row to insert, but to the server there is no distinction between these operations so it performs a PUT for both. In each case the server responds with the latest state of the row, including a row etag which the phone needs to save with its copy of the row. Since the phone has made two changes to the data on the server, it increments its data etag by two, making it seven, and saves it for the next synchronization.

![Figure 4: Example synchronization sequence between ODK Tables and ODK Aggregate.](image-url)
4.4 API Access Control Model

One design challenge of the synchronization protocol was the access control model for the HTTP API. The challenge was to make a model that would allow administrators to control access on both coarse and fine grained levels, i.e. the level of a table and the level of a row. The rest of this section describes the solution I came up with.

As mentioned before, access control at the table level is done by defining TableAcl resources on a table. You may reference Table 1 to see the elements of a TableAcl resource. A TableAcl resource consists of a scope and a role, where the scope defines which users the rule applies to and the role defines the kind of access the acl is granting. A scope has a type which is one of Default, User, or Group, and a value which is null if the type is Default, the user id if the type is User, or the group id if the type is Group. The Default scope applies to all users, a User scope applies to a specific user, and a Group scope applies to all the users within a group. To create a TableAcl, a scope is paired with a role. There are seven distinct possibilities for the role, these are outlined in Table 2.

When a user accesses a table, their actions are restricted based on the acls that have been defined. For each acl whose scope includes a certain user, that user is conferred the permissions of the role associated with that acl. That user’s effective permissions, then, are the union of permissions from all roles which apply to them.

<table>
<thead>
<tr>
<th>Role</th>
<th>Entry</th>
<th>Columns</th>
<th>Properties</th>
<th>Rows</th>
<th>Acl</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Filtered Reader</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>Filtered Writer</td>
<td>R</td>
<td>R</td>
<td>R/W</td>
<td>R/W/D</td>
<td>-</td>
</tr>
<tr>
<td>Unfiltered Reader/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filtered Writer</td>
<td>R</td>
<td>R</td>
<td>R/W</td>
<td>UR/W/D</td>
<td>-</td>
</tr>
<tr>
<td>Reader</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>UR</td>
<td>-</td>
</tr>
<tr>
<td>Writer</td>
<td>R</td>
<td>R</td>
<td>R/W</td>
<td>UR/UD</td>
<td>-</td>
</tr>
<tr>
<td>Owner</td>
<td>R/D</td>
<td>R</td>
<td>R/W</td>
<td>UR/UD</td>
<td>R/W/D</td>
</tr>
</tbody>
</table>

\[R = \text{Read}, \ W = \text{Write}, \ D = \text{Delete}, \ \UR = \text{Unfiltered Read}, \ \UW = \text{Unfiltered Write}, \ \UD = \text{Unfiltered Delete}\]

Table 2: Permissions assigned to each type of TableRole. Reads and writes to Rows are by default filtered, only roles with the unfiltered read or unfiltered write permissions have full access to the table rows.

Aside from table-level access control, the access control model also allows for row-level control. This is achieved through the filterScope property of each Row. When a Row is first created, it’s filterScope is set to null. This means that only Readers, Writers, and Owners can see the Row. Later the filterScope property can be set to allow access to users without unfiltered permissions. The filterScope is exactly the same kind of scope as seen in a TableAcl, so it will be set with a type and value which define a set of users. This becomes the filter condition for that Row, and from then on any users with filtered access to the table who fall within the filterScope are able to access the Row. For example, say there is a user and a Row in a table. If the user is a Filtered Reader on the table, and the Row has a
filterScope of Default, then the user will be able to read the Row because (a) the user has filtered read access to the table due to her role and (b) the Default scope makes the Row accessible to anyone. If, however, the filterScope was set to User with the id of some other user, then our user would not be able to access the Row because she would not fall within the scope of the filter condition. Of course, Readers, Writers, and Owners would still have unfettered access to the Row.

4.5 Phone Synchronization

Another challenge in building the synchronization protocol was designing the system for keeping track of synchronization state on the phone, specifically the synchronization state for the rows of a table. The main requirement was that the phone needs to know what changes should be propagated up to the server, and it needs to do this in a reliable, error resilient manner.

A simple, common solution is to have a 'dirty' and 'deleting' bit for each row, where the dirty bit indicates the row needs to be created or updated on the server and the deleting bit indicates it should be deleted from the server. This basic scheme works, but one problem is that it treats inserts and updates with the same dirty bit. One requirement of the synchronization protocol was that it retain the ability to be extended to other backends besides Aggregate. Some backends may need to treat inserts and updates differently, so the dirty bit must be split into an 'inserting' bit and an 'updating' bit. Another problem arises in dealing with row conflicts on the phone (conflicts will be discussed in depth in the next section, 'Conflict Resolution'). If a row is in conflict, the phone needs a mechanism for notifying the user of it, so another bit, the 'conflicting' bit, must be added to each row. With these four bits—inserting, updating, deleting, and conflicting—it soon becomes confusing when and where each of them should be set and unset. For example, when updating a row on the phone, the updating bit should get set to indicate a required update to the server, unless the inserting bit is set, in which case the updating bit should remain clear because the row has not yet been inserted. If the deleting or conflicting bits are set, it complicates matters even further.

A big simplification of these state bits was to collapse them all into a single column with an integer describing the state. This made each state mutually exclusive which in turn made it possible to reason through every possible transition between states and subsequently design a finite state machine for the life cycle of a row on the phone. This state machine made it easier to both implement the phone side of the synchronization protocol and reason about its behavior.

The state machine is shown in Figure 5. It defines six possible states for a row on the phone:

Null: the row does not exist
Inserting: the row needs to be inserted into the table 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 table on the server
Deleting: the row needs to be deleted from the table on the server
Conflicting: both the phone 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 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. If the user deletes the row then the row is simply deleted, in this case the server will never learn that the row ever existed. If the user updates the row then the row stays in the Inserting state because to the server it will still be a new row. Finally, if the synchronization process is run, then the row will be inserted into the table on the server and the synchronization process will transition the row to the Rest state to indicate all of its changes have been communicated with the server.

Figure 5: State machine for the synchronization state of a row stored in ODK Tables.

4.6 Conflict Resolution

One of the most challenging aspects of designing the synchronization protocol was to create a process for resolving conflicts created when two users make simultaneous updates to the same piece of data. The reason this was so challenging was that there were many different, equally valid approaches to solving it. This is evidenced by the fact that major mainstream technologies, like CouchDB and Google FusionTables, often take completely different approaches to dealing with conflicts resulting from concurrent updates.
To be more specific, a conflict is defined as happening whenever two users concurrently update the same row in a table. Only one version of the row can ultimately survive in the table, so there must be a mechanism in place for taking the conflicting rows and somehow merging them to produce a single row.

There are two main tradeoffs for conflict resolution strategies, these are automatic vs manual resolution and server vs client side resolution. The first tradeoff, automatic vs manual resolution, refers to whether the resolution process is driven by a decision making algorithm in the code or if it requires user intervention to proceed. The second tradeoff is about where conflicts are dealt with. In server-side conflict resolution, conflicts happen only on the server where users on the phone will never see them. Conversely, in client-side conflict resolution conflicts happen only on the phone and the server never knows of them.

In the end, I chose to go with manual, client-side conflict resolution for the synchronization protocol. Automatic resolution was deemed less desirable because it would have greatly increased the complexity of the application and it would ultimately limit the flexibility of the protocol. In an automatic resolution scheme an administrator would want to be able to configure the resolution scheme, maybe 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. This would have made the server side application much more complex to implement and it likely would still lack certain configuration possibilities that organizations might desire. So I decided on manual resolution, which is simple to implement and puts an intelligent decision maker, i.e. the user, behind the resolution process.

Client-side resolution was chosen over server-side resolution because it limits the number of conflicts possible on a single row and it puts the responsibility to resolve a conflict into the hands of the user who probably knows best how to deal with it. If server-side resolution were chosen, then there could theoretically be as many conflicting rows as there were simultaneous updates. And since conflicts are not resolved until a human resolves them, it is possible the user in charge of resolving conflicts would forget to do it and conflicts would pile up on the server. By always keeping the server in a consistent state, there can only ever 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.

5 Evaluation

In this section I evaluate the synchronization protocol. First I revisit the requirements laid out earlier and discuss how the synchronization protocol does or does not meet them. Then I discuss the results of some preliminary scalability and performance tests of the protocol.

5.1 Evaluation of Requirements

Work with ODK Tables. The synchronization protocol was designed and built using ODK Tables.

Back up data and metadata. The rows resource of a table saves data while the properties resource saves metadata.
Share data across phones and keep it synchronized. Multiple phones can retrieve and update data in a table. Synchronization of changes is managed with a form of optimistic concurrency control.

Robust to low network connectivity. The synchronization process on the phone uses Android’s built in SyncAdapter interface which includes retry logic and exponential backoff. The SyncAdapter mechanism is the same one used for GMail and other services on the phone.

Extensible to other synchronization backends. The details of the phone synchronization state and Aggregate communication are separated. There is a generalized Synchronizer interface which can be implemented to synchronize the phone with a different backend.

Authentication based on Google accounts. Authentication uses Google accounts and the OAuth2 protocol.

Support a simple mechanism for groups of users. Simple groups, which are just lists of users, may be created.

Provide moderately complex and fine-grained access control that allows both table and row-level permissions. The access control scheme allows table-level access control through the acl resource and row-level access control through the filterScope property of each Row.

Straightforward conflict resolution. Conflict resolution uses manual client-side resolution on the phone. There is only ever a single conflict between the user’s latest changes and the latest version on the server.

Simple, easy to understand HTTP API. The HTTP API is based on the ideas of REST which is a popular and well understood architecture for building web APIs.

Maintain complete history of changes to a table. The complete history of changes to a table is tracked on the server. This history is exposed in the API through the diff resource.

Easy to deploy and maintain. ODK Tables is an Android application and as such is installed using the standard Android installation process which is easy and simple. ODK Aggregate comes with a graphical installer which allows organizations to configure it and additionally provides scripts to aid with deployment.

Affordable solution for smaller organizations. Both tools are free and open source software and there are no restrictions on their usage. Organizations only need to worry about the hardware and operating costs of a project.

Scalable to moderately sized datasets and deployments. This is discussed in the next section, Performance and Scalability.

5.2 Performance and Scalability

To test the performance and scalability of the synchronization protocol, two aspects of the protocol were chosen as the most likely bottlenecks and evaluated both by their runtime and number of database operations. These two aspects were the size of a table (number of rows) and the number of concurrent accesses to a single table.

To run these tests an App Engine account running ODK Aggregate was created. Since the server is the center of the synchronization protocol architecture it is the possible bottleneck in the system, therefore a multithreaded test harness was created to make HTTP API calls
and load the server. The App Engine instance was configured with a Frontend instance class of F1, meaning 600 MHz and 128MB; Idle instances from a minimum of 4 to an Automatic maximum; and Pending latency from an Automatic minimum to an Automatic maximum. Please see the App Engine documentation [12] for more in-depth information on what the configuration parameters mean.

The first aspect tested was how the synchronization protocol scales with an increasing table size. To do this a test was run simulating a single user synchronously creating a table, adding a number of rows to it, performing one update to each row, then deleting the table. This was done multiple times for tables with 10 rows, 100 rows, and 1000 rows.

![Figure 6: Run times in milliseconds for table size tests.](image)

Figure 6 shows the run times for the table size tests while Figure 7 shows the counts of different database operations. Both the run time and total database operations grew linearly with the table size, which makes sense because adding or updating a row should be an approximately constant time operation. This means that the scalability of the dataset size, in the case of a single user, is mostly determined by what the underlying database can handle.
The second aspect tested was how the synchronization protocol scales with concurrent access to the same table. In this test, a number of users were simulated trying to concurrently put five rows each into a table. Since the whole table must be locked on each update, this created a lot of contention for the lock. On App Engine, requests are only allowed to take a certain amount of time, between about 30 and 60 seconds, and because of this, the simulated users’ requests often timed out waiting for the lock. Therefore a simple exponential backoff scheme was used to retry each request until it was successful.

This test was run with 1, 10, 20, 30, 40, and 100 simulated users. Figure 8 shows a graph of the run times for these tests and Table 3 shows the data for referencing the actual numbers. The counts of datastore operations is not included because the logging mechanism which tracks them reported obviously erroneous data, most likely due to the high degree of concurrency experienced by the application.

Looking at Figure 8, we can see that the run time scaled close to linearly as the number of users increased, perhaps a bit faster. After about 30 concurrent users the number of lock and request timeouts was fairly high, to the point where it would be significantly bother a user trying to synchronize with the server. However, if the user does not request an immediate
Figure 8: Run times in milliseconds for multiple concurrent users tests.

sync and instead lets the synchronization process on the phone run in the background at the discretion of the ODK Tables application, then this is not as big of a worry.

The final question is whether or not the synchronization protocol meets the scalability requirements laid out in the beginning, namely that it should be able to handle thousands of rows and hundreds of phones sharing a single table.

Thousands of rows appears like it would be no problem for the synchronization protocol in terms of operating correctly. However, seeing that the table size test ran on the order of 3.5 million milliseconds, or close to one hour, for a table of 1000 rows, it would be time consuming to upload a large table from ODK Tables to ODK Aggregate. Thus the creation of a web interface to the synchronization protocol on the Aggregate side would greatly facilitate setting up the initial data for a project.

For hundreds of phones sharing a single table, it depends on how many concurrent updates to a table are expected. Even in a deployment of five hundred phones there may only ever be ten simultaneous updates to the same table. In the multiple concurrent users tests, ten simultaneous updates were resolved on the order of 10,000 milliseconds, or 1.5-2 minutes, which is slow but acceptable. The test also included the time to create and delete the table,
Concurrent Users Run Time (ms) Concurrent Users Run Time (ms)
1 11600 1 20 254404
1 14805 1 20 287399
1 15915 1 20 385463
1 18809 1 20 443810
1 24103 1 30 410311
10 102262 10 30 480719
10 104120 10 30 644286
10 108688 10 40 870990
10 82464 10 40 957004
10 89735 10 100 3264205

Table 3: Run times in milliseconds for multiple concurrent users tests.

so the real time would be a bit shorter.

However, because of the table lock on each update, the synchronization protocol does not perform as well for highly concurrent access. In this case, organizations will either have to choose a different technology or partition their data into multiple smaller tables which will experience less contention.

As a final note, these tests were run with a fairly modest amount of hardware that is hopefully representative of what an organization in the developing world might be able to afford for a smaller deployment. Running all of the tests cost no more than $20 total.

6 Conclusion

With smartphones in widespread use in the developing world there is more potential for new ways to leverage technology in ICTD work. A key feature of smartphone applications in the developed world is synchronization of data between a backend server and one or more phones, but to date none of the major technologies offering this functionality meet the needs of developing world projects and organizations. To remedy this, I have designed a synchronization protocol based on existing ODK tools which allows for sharing and synchronizing tables of data between phones. It is available as open source and free software, and is flexible enough for a wide variety of applications. Additionally, its performance is well suited to scenarios with moderately sized datasets and deployments. While the synchronization protocol has not been released to the general public, the code can be found in the ODK Aggregate project at http://code.google.com/p/opendatakit/source/checkout?repo=aggregate and in the ODK Tables project at http://code.google.com/p/opendatakit/source/checkout?repo=tables.
7 References