

Towards Collaborative Editing Of Structured Data On Mobile Devices

Nicholas Palmer, Emilian Miron, Roelof Kemp, Thilo Kielmann, Henri Bal
Vrije Universiteit
Amsterdam, The Netherlands
{palmer,rkemp,kielmann,bal}@cs.vu.nl, emilian.miron@gmail.com

Abstract—The age of collaborative editing applications on mobile devices is upon us. However, such applications traditionally rely on centralized servers and thus do not operate in fully decentralized environments. This is a problem on mobile devices where network partitions are the norm due to mobility. Furthermore, such systems typically use either a fixed schema for the data, which makes them inflexible to change and leads to abuse of structured data fields for purposes other than the original intent, or else are based on XML, which limits data to document oriented data stores or else makes querying much more cumbersome for developers and users. In contrast, our Interdroid Versioned Database system provides distributed, fully decentralized, compact, relational, versioned databases for Android powered mobile phones. It offers application designers a unique set of tools for easily building decentralized collaborative applications on Android powered mobile devices using familiar ContentProvider and SQL like interfaces. Unfortunately, this system requires that the structure of the database and the user interface (UI) used to edit records in the database to be written at compile time. What users would like is to be able to define and adapt the structure of the data at runtime. What developers would like is a system which makes building structured applications, including editing UI even easier than it is with our prior work.

In this paper we present an extension to our Interdroid Versioned Database system which adds the ability to define a ContentProvider using an Avro schema, as well as a generic editing interface for instances of that schema. We demonstrate how this system allows us to create powerful data oriented applications at either compile or runtime using an example “To Do” application, and detail how this work will serve as a basis for our future work on merging shared structured data.

I. INTRODUCTION

The face of computing is changing rapidly. Computers used to be large and bulky requiring wiring to deliver both power and network connectivity and a desk to sit them on. Today, personal computers are small, battery powered, come equipped with multiple wireless networking technologies, and are usually located in our pocket. We use these pocket devices, called *smartphones*, for a multitude of purposes; from accessing multimedia content to communicate via social networks, from navigation to search, and much, much more. It is easy to see that smartphones are revolutionizing computing as we know it, and have even forged the new wave of tablet devices, and are often touted as the next generation of computing[11].

Arguably, the high utility of these devices comes from their ability to store data of interest to the user, including so called personal information like contacts, calendar, and to-do lists, but also music and video, while simultaneously having various network technologies, which allow users to synchronize, share and exchange this information via the internet. The combination of these features has given rise to

many collaborative editing applications including Wikipedia, Google Docs, GPS Track sharing sites and many, many more.

Unfortunately, many of these applications use centralized cloud based storage solutions. This is problematic because these devices are mobile, and thus they often experience changes in network connectivity, changing from an office WiFi network, to flaky 3G connectivity on the train ride home, and then back to a different WiFi network again at home, or even forgoing data networks all together when roaming in order to avoid high costs. This causes significant problems for centralized solutions for collaborative editing on mobile devices, as network connectivity to use such applications is clearly not always available. Thus we, and others[9], [10], believe that data availability is more important to users of such devices than consistency across all nodes in the system. It is well known that consistency and availability are not both possible in the face of the network partitions[3] common on mobile devices, thus, we argue that weak consistency is the only acceptable solution to the problem.

Therefore, there is a need for decentralized synchronization systems which can provide the high availability users desire in the face of network partitions while still allowing users to edit structured data. Structured data stores, like traditional SQL databases, are valuable for their familiarity to developers and the ease with which complex queries can be performed. However, for many applications, especially those for disaster management applications, the needs of data storage are not well understood when the application is first written, or indeed before the disaster strikes[6]. What is desirable, therefore, are systems which allow users and developers to easily adapt not only the structure of the data to unexpected needs but also the user interface used to edit that data.

In this paper we present our work which makes it possible to define a schema for a structured data store at runtime and generate a user interface to edit instances of that schema on mobile devices, as well as update the schema for the database. This system provides application developers and users with the ability to easily create collaborative editing applications which use structured data stores. Contributions of this paper include a system for adapting Avro schemas into SQL storage systems, and the ability to write objects to and from both the SQL storage layer and Android bundles. Additionally, we have built a generic interface engine which can generate an edit user interface for such data at runtime, allowing us to support dynamic schemas for structured data stores, with editing and storage on mobile devices. Furthermore, we demonstrate that it is possible to declare the UI simultaneously with the schema using various properties specific to the user interface builder.

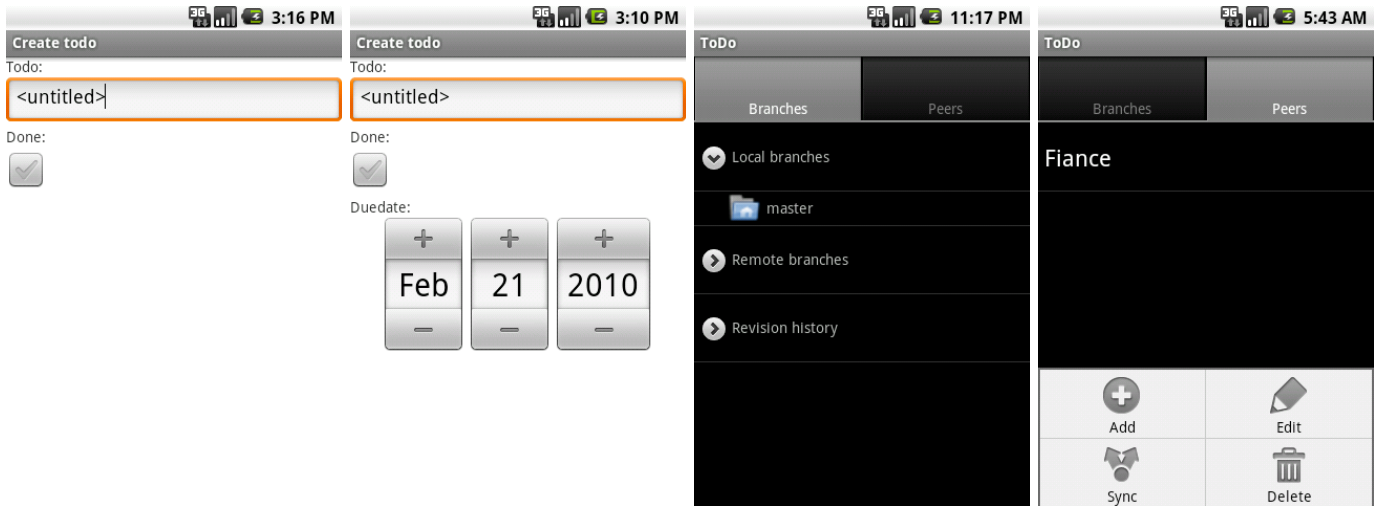


Fig. 1: Screen Shots, from left to right: a) Initial Edit UI. b) Revised Edit UI. c) Version Browser. d) Sharing Manager

We begin by motivating our system with a simple evaluation application in section II, the user interface of which is shown in Figure 1. Next, we briefly discuss the Interdroid Versioned Database system which our work extends in section III, as well as the Apache Avro¹ system we use to represent schemas in our system. We then address the design of our system in Section IV, and evaluate our work in Section V. Finally, we describe some related work in Section VI, and then conclude and describe future work in Section VII.

II. EXAMPLE APPLICATION

The best way to understand what our system is capable of doing is with a simple example. For this purpose, we have chosen a simple “To Do List” application for tracking shared tasks to be done in preparation for a wedding.

Using a schema creation application we will write as part of our Future Work, as described in Section VII, a user with a mobile device and our system begins by creating a simple schema for each to do task. The output from this application is the Avro schema shown in Figure 2.

Our runtime system uses this schema to create and instantiate a versioned database for this schema using the facilities provided by our Interdroid Versioned Database system. The system also generates the edit user interface, shown in Figure 1 a where the user can create several entries in the database. The user then commits these new entries in the versioning system, and can review the committed versions using the Version Browser activity which comes with the Interdroid Versioned

¹<http://avro.apache.org/>

```
{
  "type": "record", "name": "ToDo",
  "fields": [
    {
      "name": "todo", "type": "string",
      "default": "<untitled>"
    },
    {
      "name": "done", "type": "boolean",
      "ui.list": "false"
    }
  ]
}
```

Fig. 2: The first simple schema for a To Do application.

```
{
  "type": "record", "name": "ToDo",
  "fields": [
    {
      "name": "todo", "type": "string",
      "default": "<untitled>"
    },
    {
      "name": "done", "type": "boolean",
      "ui.list": "false"
    },
    {
      "name": "duedate", "type": "long",
      "default": "1291872831490",
      "ui.widget": "date",
      "ui.list": "false"
    }
  ]
}
```

Fig. 3: The revised schema for our To Do application.

Database system. (See Figure 1 c.)

After using the application to store some “To Do” items our groom then shares this list with his fiance by synchronizing it with her mobile phone using the Sharing Manager Activity shown in Figure 1 d.

She marks some items off the list and creates some new ones. However, in this process, she notices that what she really wants to do is be able to track when certain things on the list need to be completed by in order to keep the big day on track. So, she adds a due date field to the schema, modifying it into the schema shown in Figure 3.

Note that with our system it is possible to simultaneously declare aspects of the UI along with the schema for the field using Avro properties. This makes the system very flexible but also provides a compact way to represent the user interface and thus exchange not only the user schema for the database but also the user interface in a single, simple to edit file.

She then shares her updated database with the groom, where he notes the new version of the schema and uses it to update his database and the system generates the new user interface shown in Figure 1 b. Note that if they both edit the same entry concurrently this will generate a conflict which the system does not yet handle. We will address this limitation in our future work described in Section VII-B where we will use our UI system to merge the conflicting records. However, the system is already useful since users can read the data shared from other users.

Now that the reader has an intuitive understanding of what our system can do, we next look at the background work on which we base our system, followed by discussion of the design and implementation of this system.

III. BACKGROUND

In order to familiarize the reader with the technologies our system makes use of, we briefly discuss the role of Git, SQLite, Avro and Android Content Providers.

A. *Interdroid Versioned Database*

We base our work on the existing Interdroid Versioned Database. At the bottom of this software stack is the Versioning layer which gives us the features needed to track versions of the database and share those versions with other users. Git is an ideal basis for weak consistency replication systems, as we explain in Section III-B. The layer above this layer is the database layer, which gives us the raw data storage features of the system. Sitting on top of this layer is the ContentProvider layer which further abstracts the database and provides inter-application sharing features. Finally, the top layer consists of the user interface layer which provides activities for interacting with the versioning layer and the application itself. The versioning UI layer allows the user to browse the various versions of the database as stored in the versioning layer and launch the application in such a way that it sees the database via the content provider at a given version. It also provides user interface components for committing particular versions of the database and for sharing data, including pushing and pulling from other users or centralized repositories.

B. *Git: Distributed Version Control*

Git is a distributed version control system initially designed and developed by Linus Torvalds for Linux kernel development. In Git every working copy is a full-fledged repository with complete history and full revision tracking capability which is not dependent on network access or a central server. Git forms the basis of the versioning and sharing layer in the Interdroid Versioned Database system providing us not only with features for tracking versions and branches, but also with synchronizing with other users or centralized repositories of data via the Git push and pull protocols.

C. *SQLite: Embedded Database Management*

SQLite is an ACID-compliant embedded relational database system ideal for embedded devices such as smartphones. It is unusual in that it uses a dynamically and weakly-typed SQL syntax that does not guarantee domain integrity[1]. The overlaying ContentProvider system on Android, which we describe next, does enforce this however, so it is not an issue for our system. Furthermore, we use it to achieve a more compact on disk representation of Avro unions.

SQLite comes as part of the system level software provided by the Android platform and is the subject of many tutorials for Android. It is thus ideal for our work since it is both readily available and already familiar to developers. Important for our work however is that SQLite has limited ALTER TABLE support making it challenging for applications to adapt their database schema as needs of the application evolve. In Section IV-I, we describe how we use Avro's schema resolution to

overcome this limitation.

D. *Content Providers: Data Management on Android*

Content Providers on Android store and retrieve data across applications. The data model exposed by content providers is a table based model and queries return a Cursor object which can be used to move through the table of results and extract data. Content providers expose their data through a series of public URIs that uniquely identify the data in question on the entire device.

These URIs give application developers a simple way to deal with data on the Android platform and a large number of personal information management systems offer their data via content providers, including contacts, phone records, calendar, and many others.

E. *Avro: Schema based Serialization*

Avro is a data serialization system similar to Java Serialization. Avro provides rich data structures, a compact, fast, binary data format, a container format for persistent data stores, remote procedure call features, including integration with Apache Hadoop, and integration with dynamic languages such as Javascript. This last feature is important for our work. Alternative systems include Java Serialization, Thrift² and Protocol Buffers³. Thrift and Protocol Buffers only work with compile time code generation, whereas Avro is an object serialization system where code generation is not strictly required, though it can provide more optimized runtime performance. In this way, it is closer to Java Serialization, which inspects objects at runtime using reflection in order to generate a binary encoding of the object. Java Serialization is also known to be slow due to the reflection overhead[7]. For our system, we would like users to be able to define and modify the schema for the data at runtime and provide good performance and thus Avro is the ideal solution.

Avro relies on JSON encoded *schemas* which define the structure of the data being read or written. This allows data to be written with no per-value overhead the same as in Protocol Buffers and Thrift. However, these schemas are dynamic and can be parsed from a JSON string at runtime which allows Avro to process stored data without a code generation step. This allows Avro to do symbolic resolution of data using the schema of the read and written data which we use to handle changing schemas as described in section IV-I and will also use this for merging two databases as discussed in section VII.

IV. DESIGN AND IMPLEMENTATION

Now that the reader is familiar with the underlying technologies we base our work upon, we next discuss the design and implementation for our system and outline how it brings us closer to our vision of collaborative editing of structured data on mobile devices with malleable schemas. The system consists of two main components. The first is the Avro ContentProvider extension to the existing GenericContentProvider system provided by Interdroid Versioned Databases. This layer is responsible for parsing the Avro schema and structuring the entities required to store data of the structure represented by the schema. The other component contains user interface com-

²<http://thrift.apache.org/>

³<http://code.google.com/p/protobuf/>

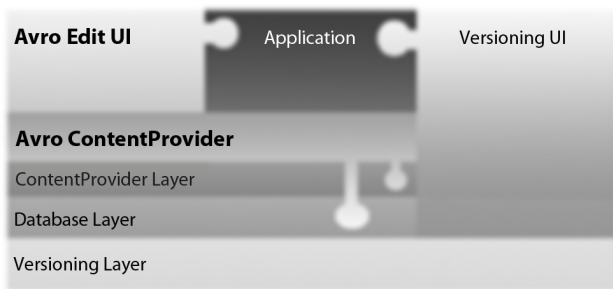


Fig. 4: Design of the Avro extensions. System components are written with black text, new components shown in bold.

ponents for listing and editing records with the given schema. These components significantly reduce the development efforts required to build the structured data editing portion of an application, but more importantly allow users to define and alter the schema for a given database at runtime. See Figure 4 for a conceptual diagram of how these components fit in to our prior work.

A. *AvroContentProvider: Avro Schema to SQL*

In order to construct an SQL schema which is capable of storing the data type represented by an Avro schema, we parse the Avro schema and convert it into an internal representation of database entities and pass this to the `GenericContentProvider` subsystem provided by the `Interdroid Versioned Database`.

The `GenericContentProvider` system can then construct an SQL database that matches the schema. For simple Avro records that only include basic types, the mapping is straight forward, with the mapping of each field in the record to a column of the correct type in the SQL schema. For more complex types, such as sub-records, unions, arrays, maps and enumerations, a more complex approach is required.

B. *Enumerations*

Avro enumerations are relatively straight forward to support. These are represented using an additional table for the values in the enumeration where the key is the ordinal in the enumeration and the value is the string representation. The SQL column for the value in the record is used as an integer foreign key into the enumeration tables ordinal column. Since the values for the enumeration are known from the schema the values for this table are added to the additional table upon creation.

C. *Arrays*

Arrays are slightly more complex. For these, an additional table is also created, as with enumerations, however they are given a complex primary key consisting of two or more columns. In the simplest case, there are two columns consisting of the key for the parent record and the index into the array. In the case of complex nestings there may be more than one primary key column for the parent type, which we will go into more details about shortly. These columns are, of course, in addition to the columns required to support the basic data type of the array.

D. *Maps*

Maps are handled the same as arrays except that instead of having the last column of the primary key be an integer index into the array, instead they are stored using the string key of the map as the key column in the table, as one would expect. They are otherwise identical to arrays. That is, arrays can be viewed as a one to many table with a primary key consisting of a reference to the, possibly complex, primary key of the parent type and an index into the array, while maps consist of a one to many table with the same references and a string for the map key.

E. *Unions*

While, records, enumerations, arrays and maps have relatively straight forward mappings, unions are a more complex and particularly interesting case. An Avro union is able to be a combination of any basic type, which can each occur only once, along with as many named types as is desired. Named types include Avro records, fixed and enums. Thus unions may not contain another union directly inside them without an intervening named record. In the case of unions we take advantage of SQLite's dynamic typing of columns. This allows us to keep the on disk representation of the data more compactly in SQL since we do not have to have separate value columns for each type a union can take on. Instead, we use a three column system for unions. The first column is a flag indicating the type of data stored in the value column. The second column contains the name of the type if the type the union is currently holding is a named type. The third column holds the actual value for the union. As an example, take the case of a union containing a string or an enumeration value. For the former, the type column will contain the flag for string, the name column will be null, since string is not a named type, and the value column will contain the actual string value the union currently holds. For the latter, the type column will contain the flag for enumeration, the name column will hold the name of the enumeration, and the value column contains an integer with a reference into the enumeration table. This gives us a compact representation of the union in a constant 3 columns, regardless of the number of branches in the union.

F. *Sub-Records*

The final type to discuss is a sub-record. Each record type is mapped to a specific table where the columns in the table correspond to the fields in the record as described above. In addition to all the fields defined in the Avro schema, each record gets an integer column used as a primary key. This key is used in two ways. First, it acts as the unique identifier for a record in the URI provided by the Content Provider. Second, it is used as the parent key in any sub-record, array or map table in order to tie the sub-type to the parent record. That is, all complex fields point to the keys of their parent records.

G. *Nested Structures*

The last bit of complexity in the SQL schema translation process is needed to deal with multiple levels of nesting within an Avro schema, for instance, an array of arrays or an array of maps. As discussed above, an array of basic types contained within a record includes the primary keys for the base record plus the offset into the array. An array of arrays therefore contains these two columns as a portion of its

primary key, along with an additional column for the index of the element in the inner array, giving a primary key consisting of three columns. This technique can handle arbitrary nesting of columns, through the use of increasingly complex primary keys.

All of this mapping is handled by our new AvroContentProvider base class which we discuss next.

The AvroContentProvider is a natural extension of Inter-droid Versioned Databases existing GenericContentProvider interface. It allows for the definition of a ContentProvider by passing an Avro schema to the constructor. What is significant about this is that this constructor can be called at runtime in order to create and initialize a new ContentProvider and underlying database based on the schema passed to the constructor.

This satisfies our requirement that the system be able to build a structured datastore at runtime from an Avro schema. Next we turn our attention to the user interface components which allow us to edit a record in this data store.

H. Application Interface

The application interface components of our system free application developers from the tedious work of writing activities to edit the structured data in their application. The framework follows the Model View Control (MVC) architectural pattern that fits naturally with Androids largely MVC orientated system. This framework handles all of the standard Android Activity lifecycle events and also integrates perfectly with the existing Versioning UI components previously described using Android's Intent system.

The model consists of a class which loads and stores the data for a particular Avro record from the database using the AvroContentProvider. This model is effectively an "Active Record[2]" pattern backed by our AvroContentProvider. It also allows changes to be buffered in memory and persisted in response to various lifecycle events routed by the control components.

The two view Activities we have added are the AvroBaseEditor for editing records and the AvroBaseList for viewing a list of records in the database. Both of these classes only require an Avro schema to build the required UI components. They both make use of an internal AvroViewFactory class which is capable of constructing the required views and sub-activities for editing or viewing a particular data record. The sub-components built by the factory can either be generic components constructed by the view factory based on the type of data being edited, or can optionally be customized using various per-field properties within the Avro schema, allowing application designers the freedom to construct custom edit widget classes, provided such classes implement the necessary interfaces for the given type. There are many other properties which we do not discuss here in the interest of brevity.

Because of the limited screen size and application memory, the actual user interface must be broken down into a number of Activities for very complex Avro schemas. For instance, in the case of sub-records, the UI needs to construct a new view Activity for editing the sub-record and displaying that on screen. This is handled by the activity by launching a new Intent and passing it the schema and ContentProvider URI for the sub-record. When editing of the sub-Record is complete,

the model of the resulting record is returned to the original Activity as a result for storage in the in memory model. In this way, editing a large and complex record is broken into a number of smaller activities which can fit in the limited memory of the device. This is necessary because the view components take up considerable memory on device and thus need to be broken up into smaller chunks which can be loaded independently.

Finally, note that it is possible to specify various aspects of the user interface in the schema directly. For the sake of brevity we do not go into details here but it is possible to hide elements, specify various widget types, and even specify custom resources for editing components, amongst others. This gives our framework considerable expressivity and we intend to expand this aspect of the system to support more styles and widgets as we discover the need for them.

The control components of this system consist of a handler for the various Android lifecycle events, as well as for constructing and controlling the in memory model of the record being edited or the list being viewed. The controller is also responsible for directing the lifecycle events received from the views to the model as required and requesting the model store itself to long term storage, in response to these events. The control components also provide various event handlers which connect the constructed UI components to the active record model the controller is using.

I. Changing Schemas

An important aspect of the system is being able to handle changes to the schema. This is done using Avro's schema resolution system. We overcome the limitation of SQLite's ALTER TABLE statement by creating a new database next to the old database which has the new schema. We then read all records in the old database using the new Avro schema and allow Avro's schema resolution features to handle the mapping. We then write each record read to the new database, and when all records have been processed, move the new database over the old database and commit it along with the new schema.

Avro's schema resolution process allows the reading of old data with a new schema provided certain rules are followed as part of the schema revision process. For example, fields can always be removed, but if a field is added, it must come with a default value. Another example is type promotion, such as promoting an integer to a long, or a long to a double, but fields can not be truncated to a smaller size. The schema editing application we will develop as part of our future work outlined in section VII-A will ensure that users follow these limits as they evolve the schema for their data.

V. EVALUATION

In order to evaluate our framework, we have implemented our simple "Wedding To Do" application described above using both the traditional Android system and our framework. The code required to instantiate the ToDoProvider for this application using our system, as with the user interface components, only requires the program to pass the schema to the constructor for the ContentProvider. This saves the programmer from considerable burden of writing a ContentProvider which tends to be very long and repetitive.

In order to test the user interface components we also construct the user interface of the “To Do” application using our new Avro based components. This only requires calling the constructor on our base class and passing it the Avro schema. Note that in both cases the only information that the user interface requires is the schema for the database in question. These two classes represent a significant abstraction of the complexity of building a user interface on Android.

While far from exhaustive, this simple application already demonstrates the power of the system we have built and represents a significant step on the road to collaborative editing of structured data.

VI. RELATED WORK

A great deal of work has been done on both synchronization and collaborative editing on mobile devices. In this section we give a far from exhaustive overview of some of the most similar systems to ours, due to space constraints.

In Syxaw[5] the authors give a general synchronization system for XML oriented documents. This system differs from ours in that we focus on structured data stores which are natural to query, while they focus on document oriented system with an emphasis on XML.

The same is true for DocX2Go[8] which makes use of optimistic replication just as our system does, and can work in a fully decentralized way. However, the XML focus of the platform makes it inappropriate for many applications. Furthermore, they do not focus on the user interface components required to edit truly structured data.

In the Disco[4] framework the authors explore how applications can handle operation of collaborative systems in the face of disconnections. This system focuses more on handling disconnection in synchronous systems and not on data representations and user interfaces, while this paper assumes asynchronous operations and focuses on schema and user interface issues.

VII. CONCLUSION

In this paper we have described our extensions to the Interdroid Versioned Database system using the Avro Schema language. This system offers users and application designers the ability to construct SQLite databases with an arbitrary schema represented using Avro’s JSON encoded schemas, at both compile and runtime. Such databases can be synchronize via Git push and pull operations. This brings the total system one step closer to the vision of allowing users to create and modify structured data systems at runtime, as well as share their data with other users. The system operates on Android powered mobile devices providing application designers the familiar ContentProvider interface to work with. We feel that the combination of features offered by this system is uniquely powerful and will enable a host of new collaborative applications.

While there is still considerable work to be done to fully enable our vision of collaborative editing of structured data on mobile devices, the current work represents a significant step towards that vision and is already very usable as is demonstrated by our “To Do” evaluation application.

A. Schema Creator Application

As a portion of our future work we have already begun work on an Avro Schema creator application. This application feeds the Schema for an Avro Schema into our existing AvroContentProvider and AvroBaseEditor in order to give a structured interface for creating Avro schemas. This application will allow users to define the schema for their data store without having to write the JSON for the schema, and will also allow a non-technical user to create the model for their data, instantiate the database and have an editing interface for their data all created at runtime. This work requires some extensions to our existing work in order to support the user interface properties available for particular field types in a way that is natural and easy for users.

B. Merging Structured Data Stores

As discussed, a portion of the motivation for this work was to enable us to generate a merge edit interface for arbitrary data. As such, the current work puts the system in very good shape to let us explore various merge and conflict resolution strategies as part of our ongoing work. We believe that because of Avro’s schema resolution facilities we will be able to support merging of conflicting databases by leveraging the schema resolution facilities of Avro to reduce conflicts and using our UI generation system in the case of conflicts the user must address.

Once all of this work is complete, we anticipate that our system will easily form the basis for a wide variety of collaborative editing applications on mobile devices.

REFERENCES

- [1] E. Codd, “Recent investigations in relational database systems,” *Data: its use, organization and management: ACM Pacific 75, Sheraton-Palace Hotel, San Francisco, April 17-18, 1975*, p. 15, 1975.
- [2] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.
- [3] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.
- [4] C. Gutwin, T. N. Graham, C. Wolfe, N. Wong, and B. de Alwis, “Gone but not forgotten: designing for disconnection in synchronous groupware,” in *CSCW ’10: Proceedings of the 2010 ACM conference on Computer supported cooperative work*. New York, NY, USA: ACM, 2010, pp. 179–188.
- [5] T. Lindholm, J. Kangasharju, and S. Tarkoma, “Syxaw: Data synchronization middleware for the mobile web,” *Mob. Netw. Appl.*, vol. 14, no. 5, pp. 661–676, 2009.
- [6] N. Palmer, “Enabling distributed applications for smart phones,” Master’s thesis, Vrije Universiteit, Amsterdam, 2007.
- [7] N. Palmer, T. Kielmann, and H. Bal, “Serialization for ubiquitous systems: An evaluation of high performance techniques on java micro edition,” in *UBICOMM*, Valencia, Spain, 2008.
- [8] K. P. Puttaswamy, C. C. Marshall, V. Ramasubramanian, P. Stuedi, D. B. Terry, and T. Wobber, “Docx2go: collaborative editing of fidelity reduced documents on mobile devices,” in *MobiSys ’10: Proceedings of the 8th international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2010, pp. 345–356.
- [9] D. Ratner, P. L. Reiher, G. J. Popek, and G. H. Kuenning, “Replication requirements in mobile environments,” *Mobile Networks and Applications*, vol. 6, no. 6, pp. 525–533, 2001.
- [10] P. Reiher, J. Popek, M. Gunter, J. Salomone, and D. Ratner, “Peer-to-peer reconciliation based replication for mobile computers,” in *European Conference on Object Oriented Programming, Second Workshop on Mobility and Replication*, 1996.
- [11] P. Zheng and L. Ni, *Smart Phone and Next Generation Mobile Computing*. Morgan Kaufmann, 2006.