Mobile Data Sync in a Blink

Nitin Agrawal, Akshat Aranya, Cristian Ungureanu
NEC Labs America

Abstract

Mobile applications are becoming increasingly data-centric – often relying on cloud services to store, share, and analyze data. App developers have to frequently manage the local storage on the device (e.g., SQLite databases, file systems), as well as data synchronization with cloud services. Developers have to address common issues such as data packaging, handling network failures, supporting disconnected operations, propagating changes, and detecting and resolving conflicts. To free mobile developers from this burden, we are building Simba*, a platform to rapidly develop and deploy data-centric mobile apps. Simba provides a unified storage and synchronization API for both structured data and unstructured objects. Apps can specify a data model spanning tables and objects, and atomically sync such data with the cloud without worrying about network disruptions. Simba is also frugal in consuming network resources.*

1 Introduction

Mobile devices are fast becoming the predominant means of accessing the Internet. For a growing user population, wired desktops are giving way to smartphones and tablets using wireless mobile networks. A recent report by Cisco [1] forecasts 66% annual growth of mobile data traffic over the next 4 years. Mobile platforms such as iOS, Android, and Windows Phone are built upon a model of local apps that work with web content. While web apps exist, a majority of smartphone usage is driven through native apps made available through their respective marketplaces. Google and Apple's marketplaces each have around 700,000 apps available [3].

A large number of mobile apps rely on cloud infrastructure for data storage and sharing. At the same time, apps need to use local storage to deal with intermittent connectivity and high latency of network access. Local storage is frequently used as a cache for cloud data, or as a staging area for locally generated data. Traditionally, mobile app developers requiring such synchronization have to roll out their own implementations, which often have similar requirements across apps: managing data transfers, handling network failures, propagating changes to the cloud and to other devices, and detecting and resolving conflicts. In a mobile marketplace targeted towards a large developer community, expecting every developer to be an expert at building infrastructure for data syncing is not ideal. Mobile developers should be able to focus on implementing the core functionality of apps.

App SDKs for both Android and iOS provide two kinds of data storage abstractions to developers: table storage for small, structured data, and file systems for larger, unstructured objects such as images and documents.

For some mobile apps it is sufficient to synchronize only structured data; for example, RSS and News Readers (FeedGoal), simple note sharing (SimpleNote), and some location-based services (Foursquare). Recent systems provide synchronized table stores [2, 4, 11] to aid such apps; iOS also provides synchronization of application's structured *Core Data* using iCloud.

For other apps, synchronization of file data alone is sufficient; for example, SugarSync, Dropbox, and Box. Services such as Google Drive and iCloud simplify data management for mobile apps requiring file synchronization.

However, of the apps that require data sync, the majority use both structured *and* object data, typically with app data (in SQLite tables) and object data such as files, cache objects, and logs (in the file system). Table 1 lists a few popular categories of such apps. As an example, apps for collaborative document editing have multiple readers and writers simultaneously editing and synchronizing the same document. Such apps require the documents and their metadata, both to be synchronized frequently and consistently; in current mobile systems, the app developer is responsible for manually handling such dependencies, making the app prone to partial data unavailability and an inefficient usage of the network.

Existing approaches to synchronization thus have several shortcomings. First, it is onerous for the app developers to maintain data in two separate services, possibly with different sync semantics. Second, even if they do, apps cannot easily build a data model that requires the table data to rely on the object data and vice versa. For example, any dependency between table and file system data will have to be handled by the app. Third, by having two separate conduits for data transfer over a wireless network, apps do not benefit from coalescing and compression to the extent possible by combining the data. To address

* Author names in alphabetical order

Application Type	Structured Data	Object Data	Example Apps
Photo Sharing	Album info, location	Images	Instagram, Gallery, Picasa
Voice Recording	Tags, timestamps	Audio files	iTalk, VoiceRecorder HD, Smart Voice
EBook Reading	Bookmarks, catalog info	MOBI, PUB files	Google Play Books, Kindle Mobile, iBooks
Video Editing	Tags, location	Raw and edited video	Magisto, iMovie, Vimeo
Music Player	Gracenote db, album info, ratings	Music files	Amazon MP3, iTunes, NPR
Document Manager	Notes, keywords, permissions	Documents, web pages	Quickoffice, Evernote, OneNote
Social Networking	News feeds, friend lists	Photos, videos	Google+, Facebook, Badoo
Continuous Sensing	Checkpoint info, sensor data	Sensor logs, snapshots	Torque, SportsTracker, Endomondo
Email	Emails, message headers, labels	Attachments	Mailbox, Outlook, Gmail

Table 1: **Synchronization of Structured and Object Data by Mobile Apps.** Table lists categories (along with examples) of popular free and paid apps that require cloud synchronization, along with the components of the apps that require structured vs. object data

these shortcomings we propose Simba, a unified table and object synchronization platform specific for mobile app development; Simba applies several optimizations to efficiently sync data over scarce network resources.

2 Background

2.1 Mobile Data Sync Services

Data synchronization for mobile devices has been studied in the past [5, 7]. Coda [7] was one of the earliest systems to motivate the problem of maintaining consistent file data for disconnected "mobile" users. Other research, particularly in the context of distributed file systems, has looked at several issues in handling data access for mobile clients, including caching [16], and weakly-consistent replication [12, 15].

A few systems provide a CRUD (Create, Read, Update, Delete) API to a synchronized table store for mobile apps. Mobius [4] and Parse [11] provide a generic table interface for single applications, while Izzy [2] (developed by us) works along multiple apps reaping additional network benefits through delay-tolerant data transfer. None of these systems support large object synchronization.

One option could be to embed large objects inside the tables of these systems. Even though such systems support binary objects (BLOBs), there is an upper limit to the size of the object that can be stored efficiently. Also, BLOBs cannot be modified in-place; objects would thus need to be split into smaller chunks and stored in multiple rows, requiring further logic to map large objects to multiple rows and manage their synchronization.

Services such as Google Drive, Box, and Dropbox are primarily intended for backup and sharing of *user* file data. Even though they provide an API for third-party apps (not just users), it only provides file sync. iCloud provides both file and key-value sync APIs, but the app still has to manage them separately.

2.2 Unifying File Systems and Databases

Simba builds upon ideas from prior work to provide a unified storage API for structured and object data. Notably, there have been several attempts to unify file systems and databases, albeit with different goals. One of the earlier works, the Inversion File System [9], uses a transactional database, Postgres, to implement a file system which provides transactional guarantees, rich queries, and fine-grained versioning. Amino [18] provides ACID semantics to a file system by using BerkeleyDB internally. TableFS [13] is a file system that internally uses separate storage pools for metadata (an LSM tree) and files (the local file system). Its intent is to provide better overall performance by making metadata operations more efficient on the disk. Recently, KVFS [14] was proposed as a file system that stores file data and file-system metadata both in a single key–value store built on top of VT-Trees, a variant of LSM trees. VT-Tree by itself enables efficient storage for objects of various sizes.

2.3 Requirements for Mobile Data Sync

While systems discussed above provide helpful insights into data sync, and in using database techniques for designing file systems, building a storage system for mobile platforms introduces new requirements. First, mobile data storage needs to be *sync friendly*. Since frequent cloud sync is necessary, and disconnected operation is often the norm, the system must support efficient means to determine changes to app data between synchronization attempts. Second, traditional file systems are not designed with mobile-specific requirements. Features such as hierarchical layout and access control are less relevant for mobile usage since data typically exists in application silos (both in iOS and Android); data sharing across apps is made possible through well-defined channels (*e.g.*, Content Providers in Android), and not via a file system.

Since the majority of user data is accessed through apps, a mobile OS needs a storage system that is more *developer-friendly* than user, with APIs that ease app development; we thus have the following design goals:

- **Easy application development:** provide app developers with a simple API for storing, sharing, and synchronizing *all* application data, structured or unstructured. The synchronization semantics should be well-defined,

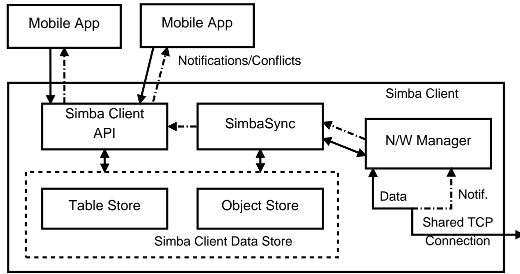

Figure 1: **Simba Client Architecture.**

even under disconnection, and if desired, should preserve atomicity of updates.

- **Sync-friendly data layout:** store app data in a manner which makes it efficient to read, query, and identify changes for synchronization with the cloud.
- **Efficient network data transfer:** use as little network resources as possible for transferring data as well as control messages (*e.g.*, notifications).

CRUD (on tables and objects)
createTable(table, schema, properties)
updateTable(table, properties)
dropTable(table)
writeData(table, table_data, object_data, atomic_sync)
updateData(table, table_data, object_data, selection, atomic_sync)
readData(table, projection, selection)
deleteData(table, selection)
Table and Object Synchronization
registerWriteSync(table, table_period, table_syncpref, object_period, object_syncpref)
unregisterWriteSync(table)
writeSyncNow(table)
registerReadSync(table, table_period, table_syncpref, object_period, object_syncpref)
unregisterReadSync(table)
readSyncNow(table)

Table 2: **Simba Client API.** Operations available to mobile apps for managing table and object data.

3 Simba Design

Simba consists of two components: a client app providing a data API to other mobile apps, and a scalable cloud store. Figure 1 shows the simplified architecture of the client, called Simba Client; it provides apps with access to their table and object data, manages a local replica of the data on the mobile device to enable disconnected operation, and communicates with the cloud to push local changes and receive remote changes. The server-side component, called Simba Cloud, provides a storage system used by the different mobile users, devices, and apps. Simba Cloud mirrors most of the client functionality and additionally provides versioning, snapshots, and deduplication. In this paper we focus on the design of the client and only discuss the server as it pertains to the client operation (Figure 1 omits the server architecture).

Simba Client is a daemon accessed by mobile apps via a local RPC mechanism. We use this approach instead of linking directly with the app to be able to manage data for all Simba-enabled apps in one central store and to use a single TCP connection to the cloud. The local storage is split into a table store and an object store (described later). SimbaSync implements the data sync logic; it uses the two stores together to determine the changes that need to be synced to the server. For downstream sync, SimbaSync is responsible for storing changes obtained from the server into the local stores. SimbaSync also handles conflicts and generates notifications through API upcalls. The Network Manager handles the network connectivity and implements the network protocol required for syncing; it also uses coalescing and delay-tolerant scheduling to judiciously use the cellular radio. Apps can individually control the maximum delay on a per-table basis; for example, apps with latency sensitive data may choose to specify a low or no delay value for certain data.

3.1 Data Model

Simba has a data model that unifies structured table storage and object storage; we chose this model to address the needs of typical cloud-dependent mobile apps. The Simba Client API allows the app to write object data and associated table data at the same time. When reading data, the app can look up objects based on queries. While permitted, objects are not required; Simba can be used for managing traditional tabular data.

Table 2 lists the Simba Client API pertaining to table management, data operations, and synchronization. For the sake of brevity, we do not discuss notifications and conflict resolution any further. There are two major goals for the API: 1) relieve the app developer from the burden of network management and data transfer 2) provide a unified logical namespace over tables and objects without the app developer having to deal with table and object storage. Note that the described API provides *one* reasonable way to express the relationship between unstructured and structured data but is not the only possible representation; the important aspect is to provide the desired I/O and sync semantics, which it does.

The first set of methods, labeled *CRUD*, are database-like operations that are popular among Android and iOS developers. In our design, we extend these calls to include object data. In our implementation, object data is accessed through the Java stream abstraction. For instance, when new rows are inserted, the app needs to provide an *InputStream* for each contained object from

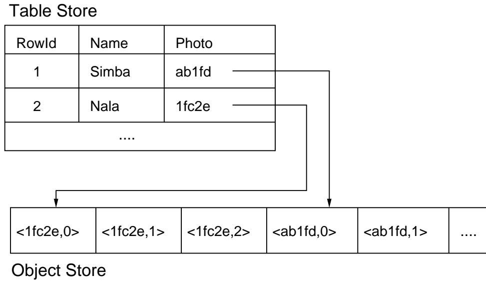

Figure 2: **Simba Client Data Store.** Table Store is implemented using a SQL database and Object Store with a key-value store based on LSM tree. Objects are split into fixed-size chunks

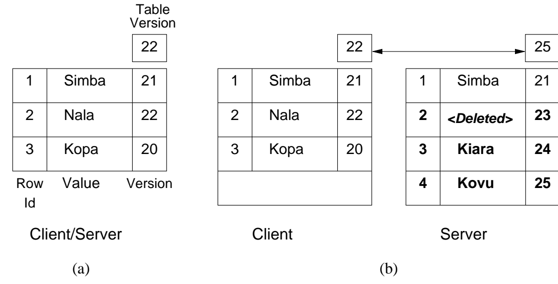

Figure 3: **Simba synchronization.** (a) Initial synchronized state. (b) Changes on the server are assigned sequential versions based on table version. During synchronization, table versions are compared to determine which changes need to be sent to client (shown here in bold).

which the data store can obtain the object data. Using streams is important for memory management; it is impractical to keep entire objects in memory. A stream abstraction for objects also allows seeking and partial reads and writes. The *writeData()* and *updateData()* always update the local store atomically, but they have an additional *atomic_sync* flag, which indicates whether the entire row set (including their objects) should be atomically synced to the cloud; we discuss this further in Section 3.3. Currently, our design allows one or more object data to be associated with each row of structured data (one-to-one or one-to-many mapping). Our current implementation does not support sharing of objects across rows (many-to-one); we will revisit this requirement in the future.

The second set of methods is used for specifying the sync policies for read (downstream) and write (upstream) sync; Simba syncs data periodically. In the downstream direction, the server uses push notifications to indicate availability of new data and Simba Client is responsible for pulling data from the cloud; if there are no changes to be synced, no notifications are sent. Table data and object data can be synced with different policies. We discuss this further in Section 3.3. *writeSyncNow()* and *readSyncNow()* allow an app to sync data on-demand.

3.2 Simba Client Data Store

The Simba Client Data Store (SDS) is responsible for storing app data on the mobile device's persistent storage. SDS needs to be efficient for storing objects of varied sizes and needs to provide primitives that are required for efficient syncing. In particular, we need to be able to quickly determine sub-object changes and sync them.

Figure 2 shows the SDS data layout. Table storage is implemented using SQLite with an additional data type representing an object identifier, which is used as a key for the object storage. Object storage is implemented using splitting objects into chunks and storing them in a key–value store that supports range queries, for example, LevelDB [8]. Each chunk is stored as a KV–pair, with the key being a $< object_id, chunk_number >$ tuple. An object's data is accessed by looking up the first chunk of the object and iterating the key-value store in key order. Splitting objects into chunks allows Simba to do network-efficient, fine-grained sync.

An LSM tree-based data structure [10] is suitable for object data because it provides log-structured writes, resulting in good throughput for both appends and overwrites; optimizing for random writes is important for mobile apps [6]. The log of the LSM tree structure is used to determine changes that need to be synced. VT-Tree [14] is a variation of LSM trees that can be more efficient; we wish to consider it in the future.

3.3 SimbaSync

Each row in Simba is a single unit of syncing. As shown in Figure 3, every table has an associated version number. Whenever a row is modified, added, or removed on the server, the current version of the table is incremented and assigned to the row. Thus, the table version is the highest version among all of its rows and no two rows have the same version (this scheme is similar to the one proposed by Renesse *et al.* [17] in the context of gossip protocols). During sync, the table versions of the client and the server are compared, and only rows having a higher version than the client's table version need to be sent to the client. Whenever a row is modified or added on the client, it is assigned a special version (-1), which marks it as a dirty row that hasn't been assigned a version yet. Once a row is synced with the server, it is assigned a real version and the client's table version is also updated to indicate that the client and the server are synced up to a particular table version.

Atomicity and sync policies: Simba supports atomic syncing of an entire row (both table and object data) over the network; this is a stronger guarantee than provided by existing sync services. We are currently investigating other forms of atomic updates, but in our prototype we do not yet provide multi-row or multi-table atomicity.

In practice, for network efficiency, mobile apps may

give up on atomic row sync. For example, a photo-sharing app that uses Simba may want to sync album metadata (*e.g.*, photo name and location) more frequently than photos, restrict photo transfer over 3G, or fetch photos only on-demand. Simba allows table and object data to have separate sync policies. A sync policy specifies the frequency of sync and the "minimum" choice of network to use. Simba also supports local-only tables (no sync), and sync-on-demand.

For downstream sync, even when different table and object sync policies are used, Simba Client can provide a consistent view of data to the app. If the object data is still unavailable or stale by the time a client app reads a row, the call will block until the object is fetched from the cloud. Similar semantics are infeasible for upstream sync since the server cannot assume client availability. However, some apps may still prefer to do non-atomic updates in the upstream direction for the sake of network efficiency/expediency; this choice is left to the app via the *atomic_sync* flag.

3.4 Writing a Simba App

We now present an example of how one would write a Simba app for Android, to show the ease of mobile app development. We take the example of a photo-sharing app that maintains name, date, and location for the photos. The app would first create the table by specifying its schema (refer to the API in Table 2).

```
client.createTable("photos", "name VARCHAR,
    date INTEGER, location FLOAT, photo OBJECT"
    , Props.FULL_SYNC);
```

The next step is to register read and write sync with appropriate parameters. In this example, the app wants to sync photo metadata every 2 minutes over any network, and photos every 10 minutes over wifi only.

```
client.registerWriteSync("photos", 120,
    ConnState.ANY, 600, ConnState.WIFI);
client.registerReadSync("photos", 120,
    ConnState.ANY, 600, ConnState.WIFI);
```

A photo can be added to the table with *writeData()*. We set *atomic_sync* to *false* so that photo metadata and the photo can be synced separately (non-atomically).

```
// get photo from camera
InputStream istream = getPhoto();
client.writeData("photos", new String[]{"name=
    Kopa","date=15611511","location=24.342","
    photo=?"},
  new InputStream[] {istream}, false);
```

Finally, a photo can be retrieved using a query:

```
ResultSet rs = client.readData("photos",
    new String[] {"photo"}, "name=Kopa");
// extract object's stream from result set
InputStream istream = rs.get(0).getColumn(0);
```

4 Conclusions

As mobile apps become more cloud-connected, app developers frequently need to synchronize data between mobile devices and the cloud. Existing solutions provide means to sync either structured or object data separately, but require the app to be responsible for consistency during sync, and for judiciously using the mobile network. We present Simba, a platform to rapidly develop and deploy data-centric mobile apps, providing a unified table and object API to ease app development. Simba provides background data synchronization with flexible policies that suit a large class of mobile apps while allowing efficient utilization of scarce network resources. We are currently developing a Simba Client Android prototype, a cloud storage system, and the network data transfer protocol for efficient synchronization; in this paper we present our early work on the Simba Client.

References

[1] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2011 – 2016. http://tinyurl.com/cisco-vni-12.

[2] N. Agrawal, A. Aranya, S. Hao, and C. Ungureanu. Building a Delay-Tolerant Cloud for Mobile Data. In *Proceedings of the IEEE MDM Conference (To Appear)*, June 2013.

[3] Google Says 700,000 Applications Available for Android. http://buswk.co/PDb2tm.

[4] B.-G. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan. Mobius: unified messaging and data serving for mobile apps. In *MobiSys '12*, 2012.

[5] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data Staging on Untrusted Surrogates. In *FAST '03*, San Francisco, CA, Apr. 2003.

[6] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. In *FAST '12*, February 2012.

[7] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1), February 1992.

[8] LevelDB: A Fast and Lightweight Key/Value Database Library. code.google.com/p/leveldb.

[9] M. A. Olson. The Design and Implementation of the Inversion File System. In *USENIX Winter '93*, San Diego, CA, Jan. 1993.

[10] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree. In *Acta Informatica*, June 1996.

[11] Parse. http://parse.com.

[12] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: a platform for content-based partial replication. In *NSDI'09*.

[13] K. Ren and G. Gibson. TABLEFS: Embedding a NoSQL database inside the local file system. In *APMRC*, November 2012.

[14] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, , J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-Trees. In *FAST '13*, February 2013.

[15] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95*, New York, NY, USA, 1995.

[16] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating Portable and Distributed Storage. In *FAST '04*, pages 227–238, San Francisco, CA, April 2004.

[17] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *LADIS '08*, pages 6:1–6:7, New York, NY, USA, 2008. ACM.

[18] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, June 2007.