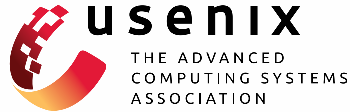

Reliable, Consistent, and Efficient Data Sync for Mobile Apps

Younghwan Go, *Korea Advanced Institute of Science and Technology (KAIST) and NEC Labs;*
Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu, *NEC Labs*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/go

This paper is included in the Proceedings of the
13th USENIX Conference on
File and Storage Technologies (FAST '15).
February 16–19, 2015 • Santa Clara, CA, USA
ISBN 978-1-931971-201

Open access to the Proceedings of the
13th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX

Reliable, Consistent, and Efficient Data Sync for Mobile Apps

Younghwan Go[†,*], Nitin Agrawal[*], Akshat Aranya[*], Cristian Ungureanu[*]
NEC Labs America[*] KAIST[†]

Abstract

Mobile apps need to manage data, often across devices, to provide users with a variety of features such as seamless access, collaboration, and offline editing. To do so reliably, an app must anticipate and handle a host of local and network failures while preserving data consistency. For mobile environments, frugal usage of cellular bandwidth and device battery are also essential. The above requirements place an enormous burden on the app developer. We built Simba, a data-sync service that provides mobile app developers with a high-level local-programming abstraction unifying tabular and object data – a need common to mobile apps – and transparently handles data storage and sync in a reliable, consistent, and efficient manner. In this paper we present a detailed description of Simba's client software which acts as the gateway to the data sync infrastructure. Our evaluation shows Simba's effectiveness in rapid development of robust mobile apps that are consistent under all failure scenarios, unlike apps developed with Dropbox. Simba-apps are also demonstrably frugal with cellular resources.

1 Introduction

Personal smart devices have become ubiquitous and users can now enjoy a wide variety of applications, or *apps* for short, running on them. Many such apps are data-centric [2] often relying on cloud-based resources to store, share, and analyze the data. In addition to the user interface and the various features, the developer of such an app needs to build the underlying data management infrastructure. For example, in order to deliver a high-quality note-taking app such as Evernote, the developers have to build a data management platform that supports rich multimedia notes, queries on data and metadata, collaboration, and offline operations, while ensuring reliability and consistency in the face of failures. Moreover, a mobile app developer needs to meet the above requirements while also being efficient with the limited resources on mobile devices such as cellular bandwidth and battery power. The better the developer handles the above-mentioned issues the more likely the app will attract and retain users.

With the rapid growth in the number and the variety of apps in the marketplace, there is a consequent demand from practitioners for *high-level abstractions* that hide the complexity and simplify the various tasks of the app developer in managing data [6, 34, 52].

Data-sync services have emerged as an aid to developers wherein an app can offload some of its data management to a third-party service such as Dropbox, iCloud, or Google Drive. While at first such services catered to end-users who want access to their files across multiple devices, more recently such services provide SDKs for apps to use directly through CRUD (Create, Read, Update, Delete) operations [15]. Sync services are built upon decades of research on distributed and mobile data sync – from foundational work on disconnected operations [30], weakly-connected replicated storage [37, 55], and version management [42], to more recent work on wide-area database replication [58], collaborative editing [46], and caching for mobile devices [57].

The principles and mechanisms of data sync by themselves are well understood, here we do not seek to reinvent them, but a data-sync service needs to achieve a dual objective in order to be valuable to mobile apps. First, it must *transparently* handle matters of reliability, consistency, and efficiency, with little involvement from the app developer, which is challenging. As the makers of Dropbox also note, providing simplicity to users on the outside can require enormous complexity and effort *under-the-hood* [24]. Second, a data-sync service must provide a data model that is beneficial to the majority of apps; while file sync is commonplace, many apps actually operate over inter-dependent structured and unstructured data [11]. A high-level data model encompassing tables and files is of great value to app developers and the transparency must apply to this data model.

A data-sync service must preserve, on behalf of the apps, the consistency between structured and unstructured data as it is stored and shared under the presence of failures. Consider the example of photo-sharing apps such as Picasa and Instagram; typically such an app would store album information in a table and the actual images on the file system or object store. In this case, the sync service needs to ensure that there will never be dangling pointers from albums to images. Since mobile apps can crash or stall frequently for a variety of reasons [10, 50], if an app is in the middle of a data operation (a local write or sync) when a failure occurs, the sync service needs to reli-

[*]Work done as part of an internship at NEC Labs

ably detect and recover to a consistent state. Recent work has shown that several data-sync services also spread corrupt data when used with desktop file systems [61, 62]. While services already exist for file [4, 27, 56] and table [15, 29, 43] data, none meet the above criteria.

To better understand how mobile apps and sync services maintain data consistency under failures, we conducted a study of popular mobile apps for Android including ones that use Dropbox, Parse, and Kinvey for data sync. Our study revealed that apps manage data poorly with data loss, corruption, and inconsistent behavior.

We thus built Simba to manage data for mobile apps, which provides a high-level abstraction unifying files and tables. The tables may contain columns of both primitive-type (string, integer, etc.) and arbitrary-sized objects, all accessible through a CRUD-like interface. For ease of adoption, the interface is kept similar to the ones already familiar to iOS and Android developers. Apps can construct a data model spanning both tables and objects and Simba ensures that all data is reliably, consistently, and efficiently synced with the server and other mobile devices.

Simba consists of an SDK for developing mobile apps, the Simba client app (sClient) for the mobile device, and the Simba cloud server (sCloud); all apps written with the Simba SDK, *Simba-apps*, communicate *only* with the local instance of sClient which serves as the proxy for *all* interaction with sCloud. In this paper, we focus on the transparency of the high-level abstraction as it affects Simba-apps and hence primarily discuss sClient; the entire Simba service is presented in greater detail elsewhere [45].

Through case studies we show how Simba enabled us to quickly develop several mobile apps, significantly increasing development ease and functionality. Simba-apps benefited greatly from sClient's failure transparency; an app written using Dropbox failed to preserve atomicity of an entire data object leading to torn updates and synced inconsistent data under failure. Benefiting from Simba's ability to programmatically incorporate delay-tolerant data transfer, Simba-apps also exhibited reduced network footprint and gave the device increased opportunity to turn off the cellular radio.

2 Study of Mobile App Reliability

We studied the reliability of some popular mobile apps and sync services (on Android) by systematically introducing failures – network disruption, local app crash, and device power loss – and observing the recovery outcome, if any. The apps in our study use both tables and files/objects, and rely on various existing services, *i.e.*, Dropbox, Parse, and Kinvey, for data sync. We setup two Android devices with identical apps and initial state. To simulate a network disruption we activated *airplane mode* and for crashes (1) manually *kill* the app, and (2) pull the battery out; the outcomes for the two crash tests do not differ and we thus list them once, as shown in Table 1.

For the network disruption tests, some apps (*e.g.*, Hiyu, Tumblr) resulted in loss of data if the sync failure was not handled immediately after reconnection. If the app (or the notification) was closed, no recovery happened upon restart. Some apps (UPM, TomDroid, Keepass2) did not even notify the user that sync had failed. As most apps required the user to manually resync after failure, this oversight led to data perpetually pending sync. Some apps exhibited other forms of inconsistency. For TomDroid, if the second device contacted its server for sync even in absence of changes, the delete operation blocked indefinitely. For Evernote, manual re-sync after disruption created multiple copies of the same note over and over.

For the crash tests, the table-only apps recovered correctly since they depended entirely on SQLite for crash consistency. However, apps with objects showed problematic behavior including corruption and inconsistency. For YouTube, even though the object (video) was successfully uploaded, the app lost the post itself. Instagram and Keepass2 both created a local partial object; Keepass2 additionally failed to recover the table data resulting in a dangling pointer to the object. Dropbox created a conflict file with a partial object (local corruption) and spread the corruption to the second device, just like Evernote.

Our study reveals that mobile apps still lose or corrupt data in spite of abundant prior research, analysis tools, and data-sync services. First, handling objects was particularly problematic for most apps – no app in our study was able to correctly recover from a crash during object updates. Second, instead of ensuring correct recovery, some apps take the easier route of disabling object updates altogether. Third, in several cases, apps fail to notify the user of an error causing further corruption. The study further motivates us to take a holistic approach for transparently handling failures inside a data-sync service and provide a useful high-level abstraction to apps.

3 App Development with Simba

3.1 Data Model and API

Data Model: Simba's data model is designed such that apps can store *all* of their data in a single, *unified*, store without worrying about how it is stored and synced. The high-level abstraction that enables apps to have a data model spanning tables and objects is called a Simba Table (sTable in short). To support this unified view of data management, Simba, under the hood, ensures that apps always see a consistent view of data stored locally, on the cloud, and other mobile devices.

The unit of client-server consistency in Simba is an *individual* row of an sTable (sRow in short) which consists of tabular data and all objects referred in it; objects are not shared across sRows. Simba provides causal consistency

	App	DM	Disruption	Recover Outcome	Crash	Recover Outcome
With Table Only	Fetchnotes *Kinvey (notes)*	T	Upd/Del note	✓ Auto resync	Upd/Del note	✓ All or nothing
	Syncboxapp *Dropbox (notes)*	T	Upd/Del note	✓ Auto resync	Upd/Del note	✓ All or nothing
	Township *Parse (social game)*	T	Background autosave	✗ App closed with data loss	Background autosave	✓ All or nothing
	UPM *Dropbox (pwd manager)*	T	Set pwd	✗ No notification	Set pwd	✓ All or nothing
	TomDroid *(notes)*	T	Upd/Del note	✗ No notification. Del blocked if other device syncs even w/o change	Upd/Del note	✓ All or nothing
	Hiyu *Kinvey (grocery list)*	T	Upd/Del item	✗ Change loss if app is closed during disruption	Upd/Del item	✓ All or nothing
Without Obj Update	Pinterest *(social n/w)*	T+O	Create pin-board	✓ Manual resync	Set/Del pin	✓ All or nothing
	Twitter *(social n/w)*	T+O	Post (re)tweet	✓ Manual resync	Post/Del tweet	✓ All or nothing
	Facebook *(social n/w)*	T+O	Post status Post comment	✓ Auto resync ✗ Comment loss if status is closed during disruption	Post/Del status	✓ All or nothing
	Tumblr *(blogging)*	T+O	Post image	✗ Post loss if notification or app closed during disruption	Post/Del image	✓ All or nothing
	YouTube *(video stream)*	T+O	Post/Del video	✓ Auto resync	Del video Post video	✓ All or nothing ✗ Post loss even for video-upload success
	Instagram *(social n/w)*	T+O	Post image Post comment	✓ Manual resync ✗ Comment loss if image is closed during disruption	Del image Post image	✓ All or nothing ✗ Partial image created locally in gallery = corruption
With Obj Update	Keepass2 *Dropbox (pwd mgr)*	O	Set/Del pwd	✗ No notification	Set pwd	✗ Password loss; partial object created locally in "kdbx" filesystem = corruption
	Dropbox *(cloud store)*	T+O	Upd/Del file	✓ Auto resync	Upd file	✗ Partial conflict file created; file corruption and spread to second client
	Evernote *(notes)*	T+O	Upd/Del note	✗ Manual sync creates multiple copies of same note	Upd note image	✗ Note image corrupted and spread to second client

Table 1: **Study of App Failure Recovery.** DM denotes the data model (T: tables only; O: objects only; T+O: app stores both tables and objects). ✗ denotes problem behavior and ✓ indicates correct handling. "Disruption" and "crash" columns list the workload for that test.

semantics with all-or-nothing atomicity over an sRow for *both* local and sync operations; this is a stronger guarantee than provided by existing sync services. An app can, of course, have a tabular-only or object-only schema, which Simba trivially supports. Since an sRow represents a higher-level, semantically meaningful, unit of app data, ensuring its consistency under all scenarios is quite valuable to the developer and frees her from writing complicated transaction management and recovery code. Figure 1 shows Simba's data model.

Simba currently does not provide atomic sync across sRows or sTables. While some apps may benefit from atomic multi-row sync, our initial experience has shown that ACID semantics under sync for whole tables would needlessly complicate Simba design, lead to higher performance overheads, and be overkill for most apps.

API: sClient's API, described in Table 2, is similar to the popular CRUD interface but with four additional features: 1) CRUD operations on tables *and* objects 2) operations to register tables for sync 3) upcalls for new data and conflicts 4) built-in conflict detection and support for resolution. Objects are written to, or read from, using a stream abstraction which allows Simba to support large objects; it also enables locally reading or writing only part of a large object – a property that is unavailable for BLOBs (binary large objects) in relational databases [38].

Since different apps can have different sync requirements, Simba supports per-table sync policies controlled by the app developer using the sync methods (*registerWriteSync* etc). Each sTable can specify a non-zero *period* which determines the frequency of change collection for sync. A *delay tolerance* (DT) value can be specified which gives an additional opportunity for data to be coalesced across apps before sending over the network; DT can be set to zero for latency-sensitive data. Even when apps have non-aligned periods, DT enables cross-app traffic to be aligned for better utilization of the cellular radio. If an app needs to sync data on-demand, it can use the *writeSyncNow()* and *readSyncNow()* methods. Simba's delay-tolerant transfer mechanism directly benefits from prior work [22, 49]. Since sync happens in the background, when new data is available or conflicts occur due to sync, apps are informed using *upcalls*. An app can begin and end a *conflict-resolution transaction* at-will and iterate over conflicted rows to resolve with either the local copy, the server copy, or an entirely new choice.

CRUD (on tables and objects)
createTable(TBL, schema, properties)
updateTable(TBL, properties)
dropTable(TBL)
outputStream[] ← writeData(TBL, TBLData, objColNames)
outputStream[] ← updateData(TBL, TBLData, objNames, selection)
inputStream[] ← rowCursor ← readData(TBL, projection, selection)
deleteData(TBL, selection)

Table and Object Synchronization
registerWriteSync(TBL, period, DT, syncpref)
unregisterWriteSync(TBL)
writeSyncNow(TBL)
registerReadSync(TBL, period, DT, syncpref)
unregisterReadSync(TBL)
readSyncNow(TBL)

Upcalls
newDataAvailable(TBL, numRows)
dataConflict(TBL, numConflictRows)

Conflict Resolution
beginCR(TBL)
getConflictedRows(TBL)
resolveConflict(TBL, row, choice)
endCR(TBL)

Table 2: **Simba Client Interface.** Operations available to mobile apps for managing table and object data. TBL refers to table name.

3.2 Writing a Simba App

Simba's unified API simplifies data management for apps; this is perhaps best shown with an example. We consider a photo-sharing app which stores and periodically syncs the images, along with their name, date, and location. First, create an sTable by specifying its schema:

```
sclient.createTable("album", "name VARCHAR, date
  INTEGER, location FLOAT, photo OBJECT", FULL_SYNC);
```

Next, register for read (download) and write (upload) sync. Here, the app syncs photos every 10 mins (600s) with a DT of 1 min (60s) for both reads and writes, selecting WiFi for write and allowing 3G for read sync.

```
sclient.registerWriteSync("album",600,60,WIFI);
sclient.registerReadSync("album",600,60,3G);
```

A photo can be added to the table with *writeData()* followed by writing to the output stream.

```
// byte[] photoBuffer has camera image
List<SCSOutputStream> objs = sclient.writeData("album"
     , new String[]{"name=Kopa","date=15611511","
     location=24.342"}, new String[] {"photo"});
objs[0].write(photoBuffer); objs[0].close();
```

Finally, a photo can be retrieved using a query:

```
SCSCursor cursor = sclient.readData("album", new
     String[] { "location", "photo" }, "name=?", new
     String[] { "Kopa" }, null);
// Iterate over cursor to get photo data
SCSInputStream mis = cursor.getInputStream().get(1);
```

4 Simba Design

4.1 Simba Server (sCloud)

The server is a scalable cloud store that manages data across multiple apps, tables, and clients [45]. It provides a network protocol for data sync, based on a model in which it is the responsibility of an sClient to pull updates from the server and push any local modifications, on behalf of all device-local Simba-apps; the sClient may register with the server to be notified of changes to subscribed tables.

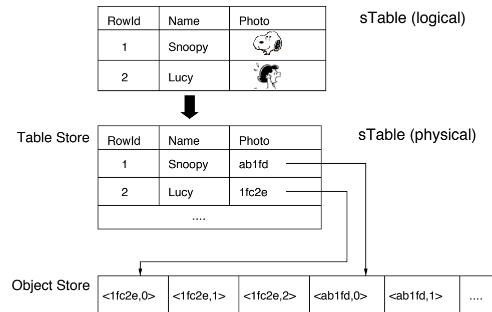

Figure 1: **Simba Client Data Store.** Table Store is implemented using a SQL database and Object Store with a key-value store based on LSM tree. Objects are split into fixed-size chunks.

Sync Protocol: To discuss sClient's design we need to refer to the semantics offered by the server through the network protocol. The server is expected to provide durability, atomicity of row updates, and multi-version concurrency control. Thus, the sClient is exposed to versions, which accompany any data in messages exchanged with the server. Simba implements a variant of *version vectors* that provides concurrency control with *causal consistency* semantics [33]. Since all sClients sync to a central sCloud, we simplify the versioning scheme to have one version number per row instead of a vector [42]. Each sRow has a unique identifier ID_{row} generated from a primary key, if one exists, or randomly, and a version V_{row}.

Row versions are incremented at the server with each update of the row; the largest row version in a table is maintained as the table version, V_{table}, allowing us to quickly identify which rows need to be synchronized. A similar scheme is used in gossip protocols [60]. Since Simba supports variable-sized, potentially large, objects, the protocol messages explicitly identify objects' partially-changed sets that need to be applied atomically.

4.2 Simba Client (sClient)

sClient allows networked Simba-apps to continue to have a local I/O model which is shown to be much easier to program for [14]; sClient insulates the apps from server and network disruptions and allows for a better overall user experience. Figure 2 shows the simplified architecture of the sClient; it is designed to run as a device-wide service which (1) provides all Simba-apps with access to their table and object data (2) manages a device-local replica to enable disconnected operations (3) ensures fault-tolerance, data consistency, and row-level atomicity (4) carries out all sync-related operations over the network. Simba-apps link with sClient through a lightweight

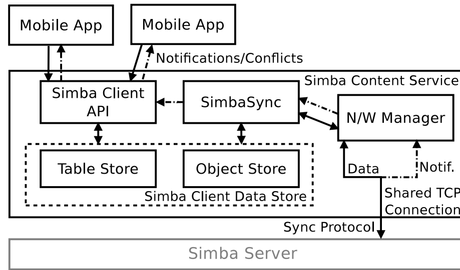

Figure 2: **Simba Client Architecture.**

library (sClientLib) which provides the Simba Client Interface (Table 2) and forwards client operations to sClient; the apps are alerted through upcalls for events (*e.g.*, new data, conflict) that happen in the background. Finally, sClient monitors liveness of apps, so that memory resources can be freed in case of app crash.

The sClient data store (§4.2.1) provides a unified abstraction over a table store and an object store. SimbaSync performs sync processing (§4.2.2) with the sCloud; for upstream sync, it collects the locally-modified data, and for downstream sync, it applies changes obtained from the server into the local store, detects conflicts, and generates upcalls to apps. The sync protocol and the local data store together provide transparent failure handling for all Simba-apps (§5). The Network Manager handles all network connectivity and server notifications for the sClient (§6); it provides an efficient utilization of the device's cellular radio through coalescing and delay-tolerance.

Implementation: sClient is currently implemented on Android, however, the design principles can be applied to other mobile platforms such as iOS. sClient is implemented as a daemon called the Simba Content Service (SCS) which is accessed by mobile apps via local RPC; on Android we use an AIDL [1] interface to communicate between the apps and the service. An alternate approach – to link directly with the app – is followed by Dropbox [16] and Parse [43] but our approach allows sClient to shape network traffic for all Simba-apps on the same device thereby benefiting from several cross-app optimizations. While the benefits of using persistent connections have been long known [35], individual apps use TCP connections in a sub-optimal manner with frequent connection establishment and teardown. sClient's design allows it to use a single persistent TCP connection to the sCloud on behalf of multiple apps; the same connection is also reused by the server for delivering notifications, providing additional savings, similar to Thialfi [9].

A misbehaving app can potentially adversely affect other Simba-apps. In practice, we believe that developers already have an incentive to write well-behaved apps to keep users satisfied. In the future, fine-grained accounting of data, similar to Android's accounting, can be built into Simba to further discourage such behavior.

4.2.1 Simba Client Data Store

The sClient Data Store (SDS) is responsible for storing app data on the mobile device's persistent storage (typically the internal flash memory or the external SD card). For Simba-apps, this means having the capability to store both tabular data and objects in a logically *unified* manner. The primary design goal for SDS is to enable, and efficiently support, CRUD operations on sRows; this requires the store to support atomic updates over the local data. Additionally, since objects are variable-sized and potentially large, the store also needs to support atomic sync of such objects. Since the store persistently stores all local modifications, a frequent query that it must efficiently support is *change detection* for upstream sync; SDS should be able to quickly determine sub-object changes. Figure 1 shows the SDS data layout.

Objects are subdivided into fixed-size chunks and stored in a key–value store (KVS) that supports range queries. The choice of the KVS is influenced by the need for good throughput for both appends and overwrites since optimizing for random writes is important for mobile apps [28]. Each chunk is stored as a KV–pair, with the key being a $\langle object_id,\ chunk_number \rangle$ tuple. An object's data is accessed by looking up the first chunk of the object and iterating the KVS in key order.

Local State: sClient maintains additional local state, persistent and volatile, for sync and failure handling. Two persistent per-row flags, $Flag_{TD}$ (table dirty) and $Flag_{OD}$ (object dirty), are used to identify locally-modified data, needed for upstream sync. To protect against partial object sync, we maintain for each row $Count_{OO}$, the number of objects opened for update. A write transaction for a row is considered closed when all its open objects are closed. Each row has two more persistent flags, $Flag_{SP}$ (sync pending) and $Flag_{CF}$ (conflict), which track its current sync state. Finally, an in-memory *dirty chunk table* (DCT) tracks chunks that have been locally modified but not yet synced. This obviates the need to query the store for these changes during normal operation.

Implementation: We leverage SQLite to implement the tabular storage with an additional data type representing an object identifier ($object_id$). Object storage is implemented using LevelDB [32] which is a KVS based on a log-structured merge (LSM) tree [40]; LevelDB meets the throughput criteria for local appends and updates. LevelDB also has snapshot capability which we leverage for atomic sync. There is no native port of LevelDB for Android so we ported the original C++ LevelDB code using Android's Native Development Kit (NDK). We use one instance of LevelDB to keep objects for all tables to ensure sequential writes for better local performance [28]. Since the local state is stored in an sRow's tabular part, SQLite ensures its consistent update.

4.2.2 Sync processing

An sClient independently performs upstream and downstream sync. The upstream sync is initiated based on the specified periodicity of individual tables, and using local state maintained in ($Flag_{TD}$, $Flag_{OD}$) to determine dirty row data; these flags are reset upon data collection. For rows with dirty objects, chunks are read one-by-one and directly packed into network messages.

Since collecting dirty data and syncing it to the server may take a long time, we used the following techniques to allow concurrent operations by the foreground apps. First, sClient collects object modifications from LevelDB snapshots of the current version. As sClient syncs a modified object only after it is closed and the local state is updated (decrement $Count_{OO}$ by 1), sClient always ensures a consistent view of sRows at snapshots. Second, we allow sClient to continue making modifications while previous sync operations are in-flight; this is particularly beneficial if the client disconnects and sync is *pending* for an extended duration. These changes set sRow's local flags, $Flag_{TD}$ or $Flag_{OD}$, for collection during the subsequent sync. For this, sClient maintains a sync pending flag $Flag_{SP}$ which is set for the dirty rows, once their changes are collected, and reset once the server indicates success. If another sync operation starts before the previous one completes, rows with $Flag_{SP}$ already set are ignored.

Downstream sync is also initiated by an sClient in response to a server notification of changes to a table. The client pulls all rows that have a version greater than the local V_{table}, staging the downstream data until all chunks of a row are received and then applying it row-by-row onto the sClient data store in increasing V_{row} order.

Conflicts on upstream sync are determined through V_{row} mismatch on the server, while for downstream by inspecting the local dirty flag of received rows. To enable apps to automatically resolve [31] or present to its users, the server-returned conflicted data is staged locally by sClient and the relevant Simba-app is notified. sClient is designed to handle conflicting updates gracefully. Conflicted rows are marked ($Flag_{CF}$) to prevent further upstream sync until the conflict is resolved. However, apps can resolve conflicts at their own convenience and can continue reading and writing to their local version of the row without sync. We believe this greatly improves the user experience since apps do not have to abruptly interrupt operations when conflicts arise.

5 Transparent Failure Handling

Mobile apps operate under congested cellular networks [13], network disruptions [20], frequent service and app crashes [10], and loss of battery [44]. Mobile OS memory management can also aggressively kill apps [12].

Failure transparency is a key design objective for sClient which it achieves through three inter-related aspects. First, the mechanism is *comprehensive*: the system detects each possible type of failure and the recovery leaves the system in a well-defined state for each of them. Second, recovery leaves the system not merely in a known state, but one that obeys *high-level consistency* in accordance with the unified data model. Third, sClient is *judicious* in trading-off availability and recovery cost (which itself can be prohibitive in a mobile environment). Barring a few optimizations (discussed in §5.2), an sClient maintains adequate local metadata to avoid distributing state with the server for the purposes of recovery [41]. sClients are stateful for a reason: it allows the sync service, having many mobile clients, which can suffer from frequent failures, and a centralized server, to decouple their failure recovery thereby improving availability.

5.1 Comprehensive & High-level Consistent

sClient aims to be comprehensive in failure handling and to do so makes the use of a state machine [53]. Each successful operation transitions sClient from one well-defined state to another; failures of different kinds lead to different *faulty* states each with well-defined recovery.

We first discuss network failures which affect only the sync operations. As discussed previously, the server response to upstream sync can indicate either success or conflict and to downstream sync can indicate either success or incompletion. Table 3(a) describes sClient's status in terms of the local sync-pending state ($Flag_{SP}$) and the relevant server response (RC_O, RC_T, RU_O, RU_T); note that only a subset of responses may be relevant for any given state. Each unique state following a network disconnection, for upstream or downstream sync, represents either a no-fault or a fault situation; for the latter, a recovery policy and action is specified sClient. Tables 3 (b) and (c) specify the recovery actions taken for failures during upstream and downstream sync respectively. The specific action is determined based on a combination of the dirty status of the local data and the server response.

Crashes affect both sync and local operations and the state of the SDS is the same whether sClient, Simba-app, or the device crash. sClient detects Simba-app crashes through a signal on a *listener* and de-allocates in-memory resources for the app. Table 4 shows the recovery actions taken upon sClient restart after a crash; for a Simba-app crash, recovery happens upon its restart.

sClient handles both network failures and crashes while maintaining all-or-nothing update semantics for sRows – in all cases, the state machine specifies a recovery action that preserves the atomicity of the tabular and object data – thereby ensuring the consistency of an app's high-level unified view; this is an important value proposition of sClient's failure transparency to mobile apps. As seen in Table 4, when an object is both dirty *and* open ($Flag_{OD}$ = 1 & $Count_{OO}$ > 0), a crash can lead to row inconsis-

Type	State Upon Network Disconnection	Implication	Recovery Policy	Action
Up stream	SP=0	No sync	Not needed	None (no-fault)
	SP=1, before `SyncUpResult`	Missed response	Reset & retry	SP←0, TD←1, if ∃DCT OD←1
	SP=1, after `SyncUpResult`(RC_O=0)	Completed	Roll forward	None (no-fault)
	SP=1, after `SyncUpResult`(RC_O=1)	Partial response	Reset & retry	See Table 3(b)
Down stream	Before `Notify`	No sync	Not needed	None (no-fault)
	After `Notify`	Sync needed	Normal operation	Send `SyncDownstream`
	After `SyncDownstream`	Missed response	Retry	Resend `SyncDownstream`
	After `SyncDownResult`(RU_O=0)	Completed	Roll forward	See Table 3(c)
	After `SyncDownResult`(RU_O=1)	Partial response	Reset & retry	See Table 3(c)

(a) Sync Failure Detection and Recovery Policy

Flags		Resp.	Recovery Action
TD	OD	RC_T	
0	0	*	Delete entry, SP←0
0	1	*	Delete entry, SP←0, TD←1, if ∃DCT OD←1
1	0	*	Delete entry, SP←0, TD←1
1	1	*	Delete entry, SP←0, TD←1, if ∃DCT OD←1

(b) Recovery action for `SyncUpstream`

Flags		Response		Recovery Action
TD	OD	RU_T	RU_O	
*	*	*	1	Delete entry, resend w/ new V_{table}: `SyncDownstream`
0	0	1	0	Update table data
1	*	1	0	Conflict on table data

(c) Recovery action for `SyncDownstream`

Table 3: **Network Disconnection: State Space for Failure Detection and Recovery.** CF= 0 for all the above states since sync is in-progress; OO is irrelevant. ∃DCT → Obj exists in DIRTYCHUNKTABLE. Delete entry → Delete row in TBLCONFLICT and corresponding object in LevelDB. TD: Table Dirty, OD: Object Dirty, SP: Sync Pending, RC_O: Response conflict for object, RC_T: Response conflict for table, RU_O: Response update for object, RU_T: Response update for table. Note TD and OD can be re-set to 1 after SP=1 since Simba allows local ops to safely proceed even when prior sync is in-progress. * indicates recovery action is independent.

tency, *i.e.*, a *torn* write. Similarly, a network disruption during an object sync can cause a *partial* sync; sClient detects and initiates appropriate *torn* recovery.

5.2 Judicious

sClient balances competing demands: on the one hand, normal operation should be efficient; on the other, failure recovery should be transparent and cheap. sClient maintains persistent state to locally detect and recover from most failures; for torn rows, after local detection, it recovers efficiently through server assistance. There are two kinds of tradeoffs it must make to keep recovery costs low.

5.2.1 Tradeoff: Local State vs. Network I/O

When sClient recovers from a crash, it can identify whether the object was dirty using $Flag_{OD}$ but it cannot determine whether it was completely or partially written to persistent storage; the latter would require recovery. $Count_{OO}$ counter enables making this determination: if it is set to zero, sClient can be sure that local data is consistent and avoid torn recovery using the server. The cost to sClient is an extra state of 4 bytes per row. However, one problem still remains: to sync this object, sClient still needs to identify the dirty chunks. The in-memory DCT will be lost post-crash and force sClient to either fetch all chunks from the server or send all chunks to the server for chunk-by-chunk comparison. sClient thus pays the small cost of persisting DCT, prior to initiating sync, to prevent re-syncing entire, potentially large, objects. Once persisted, DCT is used to sync dirty chunks after a crash and removed post-recovery. If sClient crashes before DCT is written to disk, it sends all chunks for dirty objects.

TD	OD	OO	SP	CF	Recovery action after crash (Flags)
0	0	=0	0	0	Do nothing
				1	Conflict upcall
0	0	=0	1	–	Restart `SyncUpstream` with table data and object if ∃DCT (TD←1, OD←1 if ∃DCT, SP←0)
0	0	>0	0	0	Do nothing (OO←0)
				1	Conflict upcall (OO←0)
0	0	>0	1	–	Restart `SyncUpstream` with table data and object if ∃DCT (TD←1, OD←1 if ∃DCT, OO←0, SP←0)
0	1	=0	0	0	Start `SyncUpstream` with full object
				1	Conflict upcall
0	1	=0	1	–	Restart `SyncUpstream` with full row (TD←1, SP←0) ← No information on which object is dirty
0	1	>0	0	*	Recover *Torn write* (OD←0, OO←0)
0	1	>0	1	–	Recover *Torn write* (OD←0, OO←0, SP←0)
1	0	=0	0	0	Start `SyncUpstream` with table data
				1	Conflict upcall
1	0	=0	1	–	Restart `SyncUpstream` with table data and object if ∃DCT (OD←1 if ∃DCT, SP←0)
1	0	>0	0	0	Start `SyncUpstream` with table data (OO←0)
				1	Conflict upcall (OO←0)
1	0	>0	1	–	Restart `SyncUpstream` with table data and object if ∃DCT (OD←1 if ∃DCT, OO←0, SP←0)
1	1	=0	0	0	Start `SyncUpstream` with full row
				1	Conflict upcall
1	1	=0	1	–	Restart `SyncUpstream` with full row (SP←0)
1	1	>0	0	*	Recover *Torn write* (TD←0, OD←0, OO←0)
1	1	>0	1	–	Recover *Torn write* (TD←0, OD←0, OO←0, SP←0)

Table 4: **Client Crash: State Space for Failure Detection & Recovery.** TD: Table Dirty, OD: Object Dirty, OO: Object Open Count, SP: Sync Pending, CF: Row Conflict; * indicates recovery action independent of flag; – indicates state with flag=1 is not possible

5.2.2 Tradeoff: Local I/O vs. Network I/O

If an object does have a non-zero $Count_{OO}$ post-crash, it is indeed torn. The most obvious way to recover torn rows is to never update data in-place in the SDS, but instead always write out-of-place first; once the data is successfully written, it can be copied to the final location similar to a write-ahead-log or journaling. Instead of paying the overhead during common-case operation, in this case, sClient takes assistance from Simba.

At any point in time, Simba has *some* consistent view of

Operation	Method	Throughput (MB/s)
Update	In-place	2.29 ± 0.08
	Out-of-place	1.37 ± 0.04
Read	In-place	3.94 ± 0.04
	Out-of-place	3.97 ± 0.07

Table 5: **Server-assisted Recovery.** Comparison of in-place and out-of-place local throughput with 1KB rows

that row; the client relies on this observation to either rollback or roll-forward to a consistent state. If sClient detect a local torn row during recovery, it obtains a consistent version of the row from the server; this is akin to rollback for aborted database transactions [36]. If the server has since made progress – the client in essence rolls forward. If the client is disconnected, recovery cannot proceed, but also does not prevent normal operation – only the torn rows are made unavailable for local updates. For comparison, we also implement an out-of-place SDS; as shown in Table 5, sClient is able to achieve 69% higher throughput with in-place updates as opposed to out-of-place updates for updating rows with 1KB objects.

6 Transparent Network Efficiency

Simba sync is designed to make judicious use of cellular bandwidth and device battery through a custom-built network protocol with two optimizations:

Delay tolerance and coalescing: typically, many apps run in the background as *services*, for example to send/receive email, update weather, synchronize RSS feeds and news, and update social networking. sClient is designed as a device-wide service so that sync data for multiple independent apps can be managed together and transferred through a shared persistent TCP connection. Further, Simba supports delay-tolerant data scheduling which can be controlled on a per-table basis. Delay tolerance and coalescing has two benefits. 1) Improved network footprint: allows data transfer to be clustered, reducing network activity and improving the odds of the device turning off the radio [49]. Control messages from the server are subject to the same measures. 2) Improved scope for data compression: outgoing data for multiple apps is coalesced to improve the compression [23].

Fine-grained change detection: an entire object need not be synced if only a part changes. Even though data is versioned per row, sClient keeps internal soft-state (DCT) to detect object changes at a configurable chunk level; Simba server does the same for downstream sync.

Implementation: Even though sRows are the logical sync unit, sClient's Network Manager packs network messages with data from multiple rows, across multiple tables and apps, to reduce network footprint. Simba's network protocol is implemented using Protobufs [7], which efficiently encodes structured data, and TLS for secure network communication; the current prototype uses two-way SSL authentication with client and server certificates.

7 Evaluation

We wish to answer the following two questions:

- Does Simba provide failure transparency to apps?
- Does Simba perform well for sync and local I/O?

We implemented sClient for Android interchangeably using Samsung Galaxy Nexus phones and an Asus Nexus 7 tablet all running Android 4.2. WiFi tests were on a WPA-secured WiFi network while cellular tests were run on 4G LTE: KT and LGU+ in South Korea and AT&T in US. Our prototype sCloud is setup using 8 virtual machines partitioned evenly across 2 Intel Xeon servers each with a dual 8-core 2.2 GHz CPU, 64GB DRAM, and eight 7200 RPM 2TB disks. Each VM was configured with 8GB DRAM, one data disk, and 4 CPU cores.

7.1 Building a Fault-tolerant App

The primary objective of Simba is to provide a high-level abstraction for building fault-tolerant apps. Evaluating success, while crucial, is highly subjective and hard to quantify; we attempt to provide an assessment through three qualitative means: (1) comparing the development effort in writing equivalent apps using Simba and Dropbox. (2) development effort in writing a number of Simba-apps from scratch. (3) observing failure recovery upon systematic fault-injection in sClient.

7.1.1 Writing Apps: Simba vs. Dropbox

Objective: is to implement a photo-sync app that stores album metadata and images. App_S is to be written using Simba and App_D using Dropbox. We choose Dropbox since it has the most feature-rich and complete API of existing systems and is also highly popular [56]; Dropbox provides APIs for files (`Filestore`) and tables (`Datastore`). App_S and App_D must provide the same semantics to the end-user: a consistent view of photo albums and reliability under common failures; we compare the effort in developing the two equivalent apps.

Summary: achieving consistency and reliability was straightforward for App_S taking about 5 hours to write and test by 1 developer. However, in spite of considerable effort (3 – 4 days), App_D did not meet all its objectives; here we list a summary of the limitations:

1. Dropbox does not provide any mechanism to consistently inter-operate the table and object stores.
2. Dropbox `Datastore` in-fact does not even provide row-level atomicity during sync (only column-level)!
3. Dropbox does not have a mechanism to handle torn rows and may sync inconsistent data.
4. Dropbox carries out conflict resolution in the background and prevents user intervention.

Methodology: we describe in brief our efforts to overcome the limitations and make App_D equivalent to App_S; testing was done on 2 Android smartphones – one as

Apps	Description	Total LOC	Simba LOC
Simba-Notes	"Rich" note-taking with embedded images and media; relies on Simba for conflict detection and resolution, sharing, collaboration, and offline support. Similar to Evernote [3]	4,178	367
Surveil	Surveillance app capturing images and metadata (*e.g.*, time, location) at frequent intervals; data periodically synced to cloud for analysis. Similar to iCamSpy [26]	258	58
HbeatMonitor	Continuously monitors and records a person's heart rate, cadence and altitude using a Zephyr heartbeat sensor [63]; data periodically synced to cloud for analysis. Similar to Sportstracklive [8]	2,472	384
CarSensor	Periodically records car engine's RPM, speed, engine load, etc using a Soliport OBD2 sensor attached to the car and then syncs to the cloud; similar to Torque car monitor [59]	3,063	384
SimbaBench	Configurable benchmark app with tables and objects to run test workloads	207	48
App_S	Simba-based photo-sync app with write/update/read/delete operations on tabular and object data	527	170
App_D	Dropbox-based photo-sync app written to provide similar consistency and reliability as App_S	602	–
sClient	Simba client app which runs as a background daemon on Android	11,326	–
sClientLib	Implements the Simba SDK for writing mobile apps; gets packaged with a Simba-app's `.apk` file	1,008	–

Table 6: **Lines of Code for Simba and Apps.** Total LOC counted using CLOC; Simba LOC counted manually

writer and the other as the reader. We were successful with **1**, **2** but not with **3**, **4**.

✓1. Consistency across stores: we store App_D images in `Filestore` and album info in `Datastore`; to account for dependencies, we create an extra `Datastore` column to store image identifiers. To detect file modifications, we maintain Dropbox *listeners* in App_D.

Writes: when a new image is added on the writer, the app on the reader receives separate updates for tables and files, Since Dropbox does not provide row-atomicity, it is possible for Simba metadata columns to sync before app data. To handle out-of-order arrival of images or album info prior to Simba metadata, we set flags to indicate tabular and object sync completion; when Simba metadata arrives, we check this flag to determine if the entire row is available. The reader then displays the image.

Updates: are more challenging. Since the reader does not know the updated columns, and whether any objects are updated, additional steps need to be taken to determine the end of sync. We create a separate metadata column (MC) to track changes to `Datastore`; MC stores a list of updated app-columns at the writer. We also issue sync of MC before other columns so that the reader is made aware of the synced columns. Since Dropbox does not provide atomicity over row-sync, the reader checks MC for every table and object column update.

Deletes: once the writer deletes the tabular and object columns, both listeners on the reader eventually get notified, after which the data is deleted locally.

✓2. Row-atomicity for tables+files: for every column update, `Datastore` creates a separate sync message and sends the entire row; it is therefore not possible to distinguish updated columns and their row version at sync. Atomic sync with Dropbox thus requires even more metadata to track changes; we create a separate table for each column as a workaround. For example, for an app table having one table and one object column, two extra tables need to be created in addition to MC.

For an update, the writer lists the to-be-synced tabular and object columns (*e.g.*, $\langle col1, col3, obj2\rangle$) in MC and issues the sync. The reader receives notifications for each update and waits until all columns in MC are received. In case a column update is received before MC, we log the event and revisit upon receiving MC. Handling of new writes and deletes are similar and omitted for brevity.

✗3. Consistency under failures: Providing consistency under failures is especially thorny in the case of App_D. To prevent torn rows from getting synced, App_D requires a separate persistent flag to detect row-inconsistency after a crash, along with all of the recovery mechanism to correctly handle the crash as described in §5. Since App_D also does not know the specific object in the row that needs to be restored, it would require a persistent data structure to identify torn objects.

✗4. Consistent conflict detection: Dropbox provides transparent conflict resolution for data; thus, detecting higher-level conflicts arising in the app's data model is left to the app. Since there is no mechanism to check for potential conflicts *before* updating an object, we needed to create a persistent *dirty* flag for each object in App_D. Moreover, an app's local data can be rendered unrecoverable if the conflict resolution occurs in the background with an "always theirs" policy. To recover from inconsistencies, App_D needs to log data out-of-place, requiring separate local persistent stores.

To meet **3.** and **4.** implied re-implementing the majority of sClient functionality in App_D and was not attempted.

7.1.2 Other Simba Apps

We wrote a number of Simba-apps based on existing mobile apps and found the process to be easy; the apps were robust to failures and maintained consistency when tested. Writing the apps on average took 4 to 8 hours depending on the GUI since Simba handled data management. Table 6 provides a brief description of the apps along with their total and Simba-related lines of code (LOC).

7.1.3 Reliability Testing

We injected three kinds of failures, network disruption, Simba-app crash, and sClient crash, while issuing local, sync, and conflict handling operations. Table 7 shows,

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Tab					T			T	S	S, F, R					S	S, R	S, R, F			F, R			R	R, F
Obj						C	C, O, D	C, O, D			S, D	S, D	S, D, R	S, D, T, F	S, D	S, D, R	S, D, R, F				R	R, D, F	R	R, D, F

(a) Detection. T: $Flag_{TD}$, O: $Flag_{OD}$, C: $Count_{OO}$, D: *DCT*, S: $Flag_{SP}$, R: *Server Response Table*, F: $Flag_{CF}$

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Tab					N			N	R	R, P					R	R, LR	R, LR, P			R, P			LR	LR, P
Obj						R	LR, SR	LR, SR			R	R	R, LR	R, LR, P	R	R, LR	R, LR, P				R	R, LR, P	LR	LR, P

(b) Recovery. N: *No Op*, R: *Reset*, P: *Propagate Conflict*, LR: *Local Recovery*, SR: *Server-assisted Recovery*

Table 7: **sClient Detection and Recovery.** The table shows detection and recovery policies of sClient for failure at read, write, syncup and syncdown operations. The operations are **a:** *read tab* **b:** *get obj readstream* **c:** *read obj* **d:** *read tab+obj* **e:** *write tab* **f:** *get obj writestream* **g:** *write obj* **h:** *write tab+obj* **i:** *syncup tab only* **j:** *syncupresult for tab only* **k:** *syncup obj only* **l:** *send objfrag for obj only syncup* **m:** *syncupresult for obj only* **n:** *get objfrag for obj only syncupresult* **o:** *syncup tab+obj* **p:** *syncupresult for tab+obj* **q:** *get objfrag for tab+obj syncupresult* **r:** *notify* **s:** *syncdown* **t:** *syncdownresult for tab only* **u:** *syncdownresult for obj only* **v:** *get objfrag for obj only syncdownresult* **w:** *syncdownresult for tab+obj syncdownresult* **x:** *get objfrag for tab+obj syncdownresult*.

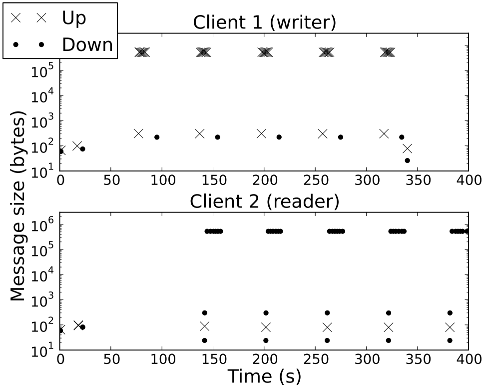

Figure 3: **Sync Network Messages.** Data and control transfer profile

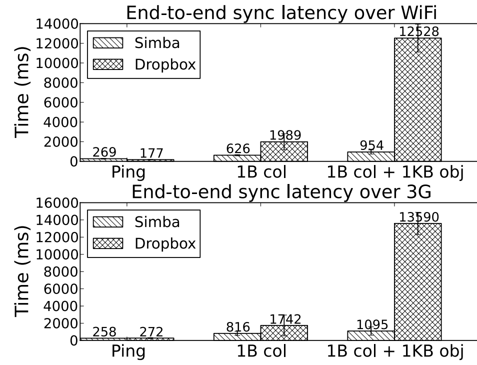

Figure 4: **Sync Network Latency.** Measured end-to-end for 2 clients

in brief, the techniques employed by sClient. For a given workload (a – x), gray cells represent unaffected or invalid scenarios, for example, read operations. A non-empty cell in detection implies that all cases were accounted for, and a corresponding non-empty cell in recovery implies corrective action was taken. The absence of empty cells indicates that sClient correctly detected and recovered from all of the common failures we tested for.

Detection: each cell in Table 7(a) lists the flags used to detect the status of tables and objects after a failure. sClient maintained adequate local state, and responses from the server, to correctly detect all failures. Change in tabular data was detected by $Flag_{TD}$ (T) for write and $Flag_{SP}$ (S) for sync as $Flag_{TD}$ is toggled at start of sync. sClient then performed a check on the server's response data (R). Sync conflict was identified by checking $Flag_{CF}$ (F). Similarly, usage of writestream and object update were detected by $Count_{OO}$ (C) and $Flag_{OD}$ (O) with the addition of DCT (D) for sync.

Recovery: each cell in Table 7(b) lists the recovery action taken by sClient from among no-op, reset, propagate, and local or server-assisted recovery. No-op (N) implies that no recovery was needed as the data was already in a consistent state. When a conflict was detected, but with consistent data, sClient propagated (P) an alert to the user seeking resolution. With the help of local state, in most cases sClient recovered locally (LR); for a torn row, sClient relied on server-assisted recovery (SR). In some cases, sClient needed to reset flags (R) to mark the successful completion of recovery or a no-fault condition.

7.2 Performance and Efficiency

7.2.1 Sync Performance

We want to verify if Simba achieves its objective of periodic sync. Figure 3 shows the client-server interaction for two mobile clients both running the SimbaBench (Table 6); on Client 1 it creates a new row with 100 bytes of table data and a (50% compressible) 1MB object every 10 seconds. Client 1 also registers for a 60-second periodic upstream sync. Client 2 read-subscribes the same table also with a 60-second period. As can be seen from the figure, the network interaction for both upstream and downstream sync shows short periodic burst of activity followed by longer periods of inactivity. Client 2's read subscription timer just misses the first upstream sync ($77s - 95s$), so the first downstream sync happens about a minute later ($141s - 157s$); for the rest of the experiment, downstream messages immediately follow the upstream ones confirming that Simba meets this objective.

We want to evaluate Simba's sync performance and

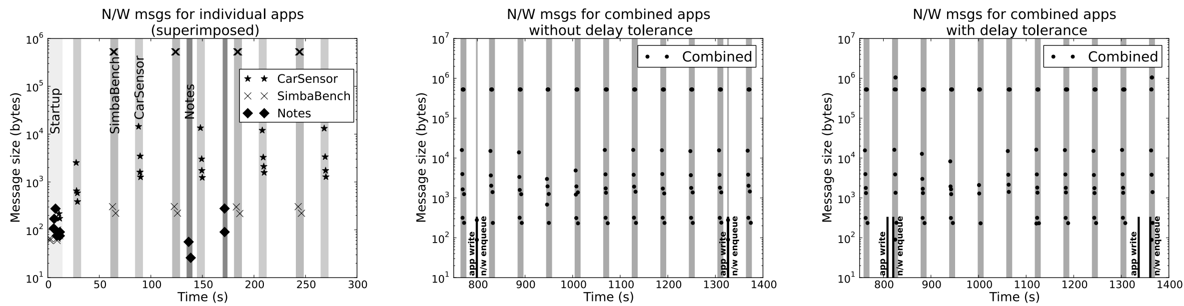

Figure 5: **Network Transfer For Multiple Apps.**

how it compares with Dropbox. Figure 4 compares the end-to-end sync latency of Simba and Dropbox over both WiFi and 4G; y-axis is time taken with standard deviation of 5 trials. For these tests we run two scenarios, both with a single row being synced between two clients: 1) with only a 1-byte column, and 2) with one 1-byte column and one 1KB object. The two clients were both in South Korea. The Dropbox server was located in California (verified by its IP address) whereas the Simba server was located on US east coast. As a baseline, we also measured the ping latency from clients to servers. Figure 4 shows that the network latency ("Ping") is a small component of the total sync latency. For both the tests, Simba performs significantly better than Dropbox; in case 1), by about 100% to 200%, and in case 2) by more than 1200%. Since Dropbox is proprietary we not claim to fully understand how it functions; it very well might be overloaded or throttling traffic. The experiment demonstrates that Simba performs well even when giving control of sync to apps.

We want to test how quickly Simba resolves conflicts for a table with multiple writers. Figure 6 shows this behavior. The x-axis shows the number of clients (min. 2 clients needed for conflict) and the y-axis shows the average time to converge (sec) and standard deviation over 5 trials. For "theirs", the server's copy is chosen *every time* and hence no changes need to be propagated back; for "mine", the local copy is chosen every time and re-synced back to the server. The "no conflicts" case is shown to establish a baseline – a normal sync still requires changes to be synced to the server; "mine" always and "theirs" always represent the worst-case and the best-case scenarios respectively with typical usage falling somewhere in between. The figure shows that for a reasonable number (*i.e.*, 5) of collaborating clients, as the number of conflict resolution rounds increases, it does not impose a significant overhead compared to baseline sync, even when selecting the server's copy; when selecting the local copy, conflict resolution is fairly quick.

7.2.2 Network Efficiency

We want to evaluate Simba's impact on network efficiency. Three apps were chosen for this experiment that generate data periodically: CarSensor app in replay mode generating about 250 byte rows every second, SimbaBench set to create 1MB rows (50% compressible) every 10s, and an app that simulates the behavior of Simba-Notes, by generating ~300 byte of data using Poisson distribution with a mean value of 300s and using a fixed seed for random number generation. CarSensor and SimbaBench run with a periodic upstream sync of 60s.

Figure 5 shows a scatter plot of the data transfer profile of the apps; y-axis is message size on a log scale, and x-axis is time in seconds. The colored bands are meant to depict temporal clusters of activity. The "Startup" band shows the one-time Simba authentication and setup, and sync registration messages for the tables. We ran the Simba apps (a) individually, (b) concurrently with Simba-Notes's DT=0, and (c) concurrently with Simba-Notes's DT=60s. Figure 5(a) shows the super-imposition of the data transfer profile when the apps were run individually, to simulate the behavior of the apps running without coordination. As also seen in the figure, while it is possible for uncoordinated timers to coincide, it is unlikely; especially so when the period is large compared to the data transfer time. Aperiodic apps like Simba-Notes also cause uncoordinated transfers. Uncoordinated transfers imply frequent radio activity and energy consumed due to large tail times. In Figure 5(b), all apps are run concurrently. The events generated by Simba-Notes are annotated. We see that the network transfers of CarSensor and SimbaBench are synchronized, but Simba-Notes still causes network transfer at irregular times (the thin bands represent network transfers by Simba-Notes). In Figure 5(c), we run an experiment similar to (b) but this time Simba-Notes employs a delay tolerance of 60s; its network activity is delayed until the next 60s periodic timer along with all pending sync activity (notice the absent thin bands). The resulting data transfer is clustered, increasing the odds of the radio being turned off. The x-axes in (b) and (c) start around 800s as we measured after a few minutes of app start.

7.2.3 Local I/O Performance

Our objective is to determine whether sClient's local performance is acceptable for continuous operation, especially since storage can be a major contributor to performance of mobile apps [28]. SimbaBench issues writes,

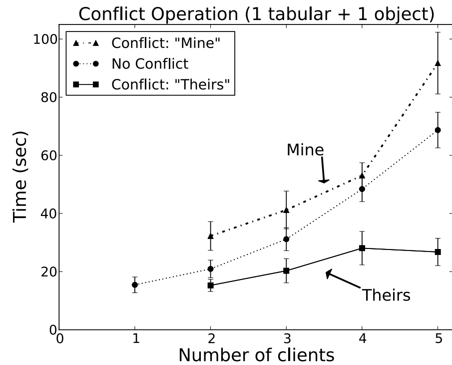

Figure 6: **Time Taken for Conflict Convergence.**

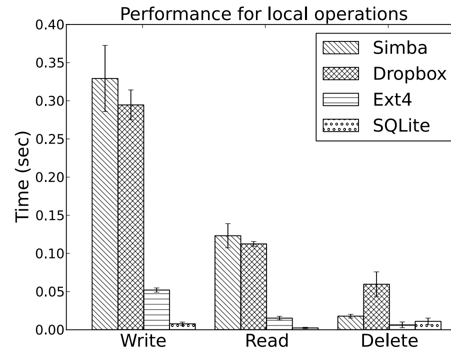

Figure 7: **sClient Local I/O Performance.**

reads, and deletes for one row of data containing one 1MB object for both sClient with Dropbox (Core API). Figure 7 shows average times and standard deviation over 5 trials; sClient is about 10% slower than Dropbox for both writes and reads, primarily due to IPC overhead as sClient is a background service on Android while Dropbox directly accesses the file system. sClient performs better for deletes through lazy deletion – data is only marked as deleted but physically removed only after sync completion. sClient and Dropbox both perform several additional operations over Ext4 and SQLite; we provide this comparison only as a baseline.

8 Related Work

Data sync and services: sync has been much studied in the context of portable devices including seminal work on disconnected operations [30], weakly-consistent replicated storage [37, 55], and data staging [18, 57].

In terms of failure transparency, Bayou [55] provides a limited discussion of its crash recovery through a write log but it does not handle objects. LBFS [37] atomically commits files on writeback, preventing corruption on crash or disruption, but does not handle tables. We find that for most apps, handling the dependencies – between tabular and object data – is the biggest source of inconsistency.

Of the existing services, Dropbox is the most comprehensive but still does not support sync atomicity for objects and tables, breaking failure transparency for several fault conditions. iCloud also provides separate mechanisms for a key-value interface and file sync. Mobius [14] provides a CRUD API to a table-sync store but does not support objects at all. Similar to Simba, Parse [43] and Kinvey [29] are mobile backend-as-a-service offering GUI integration, administration, and limited data management; they only support tables and provide last-writer-wins semantics which is inadequate for many apps. No sync service provides delay-tolerant transfer.

Fault tolerance: ViewBox [62] integrates a desktop FS with a data-sync service so as to sync only consistent views of the local data; the paper also shows how Dropbox spreads local file corruption which ViewBox addresses through checksums. Simba focuses on providing transparent fault-handling to apps; while ViewBox works only for files, Simba spans both files and tables.

Storage unification: prior work for desktop file systems has considered database integration but without network sync or a unified API. InversionFS [39] uses Postgres to implement a file system with transactional guarantees and fine-grained versioning. TableFS [51] uses separate storage pools for metadata (an LSM tree) and files to improve its own performance through metadata operations. KVFS [54] 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, which enable efficient storage for objects of various sizes; VT-Trees can be used to build a better-performing sClient data store, in the future.

Mobile data transfer: Recent research has characterized and optimized data transfer for mobile environments [21, 25, 47], especially the adverse effects of small, sporadic transfers [17, 48]; SPDY [5] extends HTTP for better compression and multiplexes requests over a single connection to save round trips. This large body of networking research has inspired Simba's network protocol.

9 Conclusions

Building high-quality data-centric mobile apps invariably mandates the developer to build a reliable and efficient data management infrastructure – a task for which few are well-suited. Mobile app developers should not need to worry about the complexities of network and data management but instead be able to focus on what they do best – implement the user interface and features – and deliver great apps to users. We built Simba to empower developers to rapidly develop and deploy robust and efficient mobile apps; through its mobile client daemon, sClient, it provides background data sync with flexible policies that suit a large class of mobile apps while transparently handling failures and efficiently utilizing mobile resources. We plan to release Simba's source code; please check with the contact author (Nitin Agrawal) for further details.

10 Acknowledgements

We thank our FAST reviewers and shepherd, Jason Nieh, for their valuable feedback. We thank Dorian Perkins for his work on Simba Cloud and the IST group at NEC Labs for its setup; Simba Cloud was also evaluated using NMC PRObE [19]. Younghwan thanks the ICT R&D program of MSIP/IITP, Republic of Korea (14-911-05-001).

References

[1] Android Developers Website. `http://developer.android.com/index.html`.

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

[3] Evernote App. `http://evernote.com`.

[4] Google Drive. `https://developers.google.com/drive/`.

[5] Google SPDY. `https://developers.google.com/speed/spdy`.

[6] Onavo. `www.onavo.com`.

[7] Protocol Buffers. `http://code.google.com/p/protobuf`.

[8] SportsTrackLive Mobile App. `http://www.sportstracklive.com/help/android`.

[9] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: a client notification service for internet-scale applications. In *SOSP '11*.

[10] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. Diagnosing mobile applications in the wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 22. ACM, 2010.

[11] N. Agrawal, A. Aranya, and C. Ungureanu. Mobile data sync in a blink. In *HotStorage '13*, San Jose, California.

[12] Android Developers. Processes and Threads. `http://developer.android.com/guide/components/processes-and-threads.html`.

[13] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *MobiSys '10*, pages 209–222, 2010.

[14] 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.

[15] Dropbox. Dropbox Datastore API. `dropbox.com/developers/datastore`, July 2013.

[16] Dropbox Sync API. `dropbox.com/developers/sync`.

[17] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *IMC '10*, 2010.

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

[19] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

[20] Y. Go, Y. Moon, G. Nam, and K. Park. A disruption-tolerant transmission protocol for practical mobile data offloading. In *MobiOpp'12*, 2012.

[21] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *MobiSys '13*, pages 153–166, New York, NY, USA, 2013. ACM.

[22] S. Hao, N. Agrawal, A. Aranya, and C. Ungureanu. Building a Delay-Tolerant Cloud for Mobile Data. In *IEEE MDM*, June 2013.

[23] D. Harnik, R. Kat, O. Margalit, D. Sotnikov, and A. Traeger. To Zip or not to Zip: Effective Resource Usage for Real-Time Compression. In *FAST 2013*, Feb 2013.

[24] D. Houston. Dropbox dbx: Developer conference keynote. `http://vimeo.com/70089044`, 2014.

[25] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An in-depth study of lte: Effect of network protocol and application behavior on performance. In *SIGCOMM '13*, 2013.

[26] iCamSpy App. Audio Video Surveillance CCTV. `http://www.icamspy.com/`.

[27] iCloud for Developers. `developer.apple.com/icloud`.

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

[29] Kinvey. `http://kinvey.com`.

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

[31] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX ATC '95*, Berkeley, CA, USA, 1995.

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

[33] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. A short primer on causal consistency. *USENIX ;login magazine*, 38(4), Aug. 2013.

[34] Z. Miners. Dropbox adds new tools to make syncing smarter. `http://www.pcworld.com/article/2043980/dropbox-adds-new-tools-to-make-syncing-smarter.html`, July 2013.

[35] J. C. Mogul. The case for persistent-connection http. In *SIGCOMM*, pages 299–313, 1995.

[36] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[37] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 174–187, Lake Louise, Alberta, Oct. 2001.

[38] MySQL. MySQL BLOB and TEXT types. http://dev.mysql.com/doc/refman/5.0/en/string-type-overview.html.

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

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

[41] J. K. Ousterhout. The role of distributed state. In *In CMU Computer Science: a 25th Anniversary Commemorative*, page pp. ACM Press, 1991.

[42] D. S. Parker Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, (3):240–247, 1983.

[43] Parse. parse.com.

[44] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *EuroSys '12*, pages 29–42, New York, NY, USA, 2012. ACM.

[45] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. Madhyastha, and C. Ungureanu. Simba: Tunable End-to-End Data Consistency for Mobile Apps. In *Proceedings of the European Conference on Computer Systems (EuroSys '15)*, Bordeaux, France, April 2015.

[46] 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*, 2010.

[47] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Web caching on smartphones: ideal vs. reality. In *MobiSys '12*, 2012.

[48] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Periodic transfers in mobile applications: network-wide origin, impact, and optimization. In *WWW '12*, 2012.

[49] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-delay tradeoffs in smartphone applications. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 255–270, New York, NY, USA, 2010. ACM.

[50] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *ACM MobiSys*, 2014.

[51] K. Ren and G. Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *USENIX ATC*, June 2013.

[52] D. ROWINSKI. Why the facebook-parse deal makes parse's rivals very, very happy. http://readwrite.com/2013/04/29/parse-acquisition-makes-its-rivals-very-happy, April 2013.

[53] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[54] 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.

[55] 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*, 1995.

[56] The Cristian Science Monitor. Dropbox has hit the 175-million-user mark, cofounder says. http://tinyurl.com/mlz8x3c, July 2013.

[57] 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.

[58] N. Tolia, M. Satyanarayanan, and A. Wolbach. Improving mobile database access over wide-area networks without degrading consistency. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 71–84. ACM, 2007.

[59] Torque App. Engine Performance and Diagnostic Tool. http://torque-bhp.com/.

[60] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. Efficient Reconciliation and Flow Control for Anti-entropy Protocols. In *LADIS '08*, 2008.

[61] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. *-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, California, June 2013.

[62] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proceedings of the 12th Conference on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[63] Zephyr. Zephyr HxM BT Heartrate Monitor. http://tinyurl.com/zephyr-sensor.