Simba: Tunable End-to-End Data Consistency for Mobile Apps

Dorian Perkins[∗†], Nitin Agrawal[†], Akshat Aranya[†], Curtis Yu[∗],
Younghwan Go[⋆†], Harsha V. Madhyastha[‡], Cristian Ungureanu[†]

NEC Labs America[†], UC Riverside[∗], KAIST[⋆], and U Michigan[‡]

Abstract

Developers of cloud-connected mobile apps need to ensure the consistency of application and user data across multiple devices. Mobile apps demand different choices of distributed data consistency under a variety of usage scenarios. The apps also need to gracefully handle intermittent connectivity and disconnections, limited bandwidth, and client and server failures. The data model of the apps can also be complex, spanning inter-dependent structured and unstructured data, and needs to be atomically stored and updated locally, on the cloud, and on other mobile devices.

In this paper we study several popular apps and find that many exhibit undesirable behavior under concurrent use due to inadequate treatment of data consistency. Motivated by the shortcomings, we propose a novel data abstraction, called a *sTable*, that unifies a tabular and object data model, and allows apps to choose from a set of distributed consistency schemes; mobile apps written to this abstraction can effortlessly sync data with the cloud and other mobile devices while benefiting from end-to-end data consistency. We build Simba, a data-sync service, to demonstrate the utility and practicality of our proposed abstraction, and evaluate it both by writing new apps and porting existing inconsistent apps to make them consistent. Experimental results show that Simba performs well with respect to sync latency, bandwidth consumption, server throughput, and scales for both the number of users and the amount of data.

1. Introduction

Applications for mobile devices, or apps, often use cloud-based services to enable sharing of data among users and to offer a seamless experience across devices; ensuring consistency across users and devices is thus a primary requirement for mobile apps. However, in spite of significant advances on distributed consistency [29, 30, 32, 55] in recent times, application developers continue to struggle with writing applications that are correct and consistent [11].

The root cause for the undesirable status quo is that every developer re-solves the same set of underlying challenges for data consistency. Instead, in this paper, we advocate the need for a high-level programming abstraction that provides *end-to-end* data consistency and, in particular, is well-suited for the needs of mobile apps. Three key properties distinguish the associated challenges from prior work.

First, unlike recent efforts that have examined when it is reasonable to trade-off consistency in order to minimize latencies in the setting of geo-distributed services [29, 30, 32, 55], whether to sacrifice strong consistency is not a choice for mobile apps; the limited and intermittent connectivity on mobile devices makes it a must for mobile apps to deal with weaker forms of consistency in order to facilitate app usage in disconnected mode. Often, mobile apps maintain a local copy of application state and user data (*e.g.*, photos, notes, and documents) and orchestrate the synchronization of these across devices in a reliable, consistent, and efficient manner.

Second, unlike prior work on managing data consistency across weakly-connected clients [26, 35, 56], for efficiency purposes, mobile apps do not always need the strongest form of consistency that is feasible even when connected. Due to carrier bandwidth caps and users' expectations of an interactive app experience, mobile app developers have to often explicitly design their apps to make do with weaker consistency. Moreover, an app typically operates on different kinds of data, which require or benefit from different forms of consistency (*e.g.*, app-crash log vs. user's shopping cart).

Third, as shown by recent studies [10, 51], the majority of mobile apps employ data models spanning databases, file sytems, and key-value stores, and it is crucial to manage data at application-relevant granularity. Cloud-connected mobile apps need to manage this inter-dependent data not only locally, but also on cloud storage and across devices.

To understand how developers manage these challenges associated with ensuring data consistency across devices, we studied a number of popular mobile apps. We make four primary observations from our study. First, we find that different mobile apps — even different components of the same app — operate with different consistency semantics; thus, there is no one-size-fits-all solution for cloud-based data sync. Second, while many apps chose weaker consistency semantics for the sake of availability and efficiency, their data synchronization is designed poorly, leading to inconsistencies and loss of user data. Third, though several apps use existing commercial sync services (*e.g.*, Dropbox), even

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/2741948.2741974

such apps exhibit undesirable behavior under concurrent use. The dominant reason being that apps choose the sync service oblivious to the ramifications of their last-writer-wins semantics on the data model. Fourth, while most apps store inter-dependent unstructured (*e.g.*, photos and videos) and structured data (*e.g.*, album info and metadata), existing sync services offer either file-only [3, 5, 25, 53] or table-only [15, 23, 42] abstractions. As a result, atomicity of granular data is not preserved under sync even for a popular app such as Evernote which claims so [16].

Mobile apps are ultimately responsible for the user experience and need to guarantee data consistency in an end-to-end manner [48]. Motivated by the observations from our study, we propose a novel data synchronization abstraction that offers powerful semantics for mobile apps. Our proposed abstraction, which we call sTable, offers tunable consistency and atomicity over coarse-grained inter-dependent data, while gracefully accommodating disconnected operations. sTables span both structured and unstructured data (i.e., database tables and objects), providing a data model that *unifies* the two. Thus, sTables enable apps to store all of their data in a single synchronized store. To our knowledge, our sTable abstraction is the first to provide atomicity guarantees across unified tabular and object rows. Furthermore, to support varying consistency requirements across apps and across an app's components, every sTable can be associated with one of three consistency semantics, resembling *strong*, *causal*, and *eventual* consistency. The sTable abstraction subsumes all network I/O necessary for its consistent sync. sTables are thus especially suited to developing cloud-connected mobile apps.

To demonstrate the simplicity and power of sTables, we have built Simba, a data-sync service that meets the diverse data management needs of mobile apps. By providing a high-level data abstraction with tunable consistency semantics, and an implementation faithful to it, Simba alleviates the deficiencies in existing data-sync services. The key challenges that Simba addresses are end-to-end atomic updates, locally and remotely, to unified app data, and an efficient sync protocol for such data over low-bandwidth networks.

We have evaluated Simba by developing a number of mobile apps with different consistency requirements. Our experimental results show that 1) apps that use Simba perform well with respect to sync latency, bandwidth consumption, and server throughput, and 2) our prototype scales with the number of users and the amount of data stored, as measured using the PRObE Kodiak and Susitna testbeds [19]. We have also taken an existing open-source app exhibiting inconsistent behavior and ported it to Simba; the port was done with relative ease and resulted in a consistent app.

2. Study of Mobile App Consistency

We studied several popular mobile apps to understand how they manage data along two dimensions: consistency and granularity. Data granularity is the aggregation of data on which operations need to apply together, *e.g.*, tabular, object, or both; in the case of cloud-connected apps, granularity implies the unit of data on which local *and* sync operations are applied together. We adopted this terminology from Gray *et al.* [21], who present an excellent discussion on the relationship between consistency and granularity.

We chose a variety of apps for our study, including: 1) apps such as Pinterest, Instagram, and Facebook that roll their own data store, 2) apps such as Fetchnotes (Kinvey), Township (Parse), and Syncboxapp (Dropbox) that use an existing data platform (in parenthesis), and 3) apps such as Dropbox, Evernote, and Google Drive that can also act as a sync service with APIs for third-party use. The apps in our study also have diverse functionality ranging from games, photo sharing, and social networking, to document editing, password management, and commerce.

2.1 Methodology

We setup each app on two mobile devices, a Samsung Galaxy Nexus phone and an Asus Nexus tablet, both running Android 4.0+, with the same user account; the devices connected to the Internet via WiFi (802.11n) and 4G (T-Mobile). We manually performed a series of operations on each app and made a qualitative assessment.

For consistency, our intent was to observe as a *user* of the system which, in the case of mobile apps, broadened the scope from *server only* to server *and* client. Since most of the apps in the study are proprietary, we did not have access to the service internals; instead, we relied on user-initiated actions and user-visible observations on mobile devices.

For granularity, we inspected the app's local schema for dependencies between tabular and object data. Apps typically used either a flat file or a SQLite database to store pointers to objects stored elsewhere on the file system. To confirm our hypothesis, we traced the local I/O activity to correlate accesses to SQLite and the file system.

First, for each app, we performed a number of user operations, on one or both devices, while being online; this included generic operations, like updating the user profile, and app-specific operations, like "starring/unstarring" a coupon. Second, for the apps supporting offline usage, we performed operations while one or both devices were offline. We then reconnected the devices and observed how offline operations propagated. Finally, we made updates to shared data items, for both offline and online usage, and observed the app's behavior for conflict detection and resolution. These tests reflect typical usage scenarios for mobile apps.

2.2 Classification

Our assessments are experimental and not meant as a proof; since automatic verification of consistency can be undecidable for an arbitrary program [44], our findings are evidence of (in)consistent behavior. We chose to classify the apps into three bins based on the test observations: *strong*, *causal*,

	App, *Function* & Platform	**DM**	**Tests**	**User-visible Outcome**	**Reasoning**	**CS**
Use Existing Platforms	Fetchnotes, *shared notes* Kinvey	T	Ct. Del/Upd on two devices	Data loss, no notification; hangs indefinitely on offline start	**LWW → clobber**	E
	Hipmunk, *travel* Parse	T	Ct. Upd of "fare alert" settings	Offline disallowed; sync on user refresh	LWW → correct behavior	E
	Hiyu, *grocery list* Kinvey	T	Ct. Del/Upd w/ both offline; w/ one user online	Data loss and corruption on shared grocery list → double, missed, or wrong purchases	**LWW → clobber**	E
	Keepass2Android⋆, *password manager* Dropbox	O	Ct. password Upd w/ both online; repeat w/ one online one offline	Password loss or corruption, no notification	**Arbitrary merge/overwrite → inconsistency**	C
	RetailMeNot, *shopping* Parse	T+O	Ct. "star" and "unstar" coupons	Offline actions discarded; sync on user refresh	LWW → correct behavior	E
	Syncboxapp⋆, *shared notes* Dropbox	T+O	Ct. Upd w/ both online; w/ one offline one online; w/ one offline only	Data loss (sometimes)	FWW; FWW (offline discarded); Offline correctly applied	C
	Township, *social game* Parse	T	Ct. game play w/ background auto-save	Loss & corruption of game state, no notification; loss/misuse of in-app purchase; no offline support	**LWW → clobber**	C
	UPM⋆, *password manager* Dropbox	O	Ct. password Upd w/ both online; repeat w/ one online one offline	Password loss or corruption, no notification	**Arbitrary merge/overwrite → inconsistency**	C
Roll-their-own platform	Amazon, *shopping*	T+O	Ct. shopping cart changes	Last quantity change saved; Del overrides quantity change if first or last action	**LWW → clobber**	S+E
	ClashofClans, *social game*	O	Ct. game play w/ background auto-save	Usage restriction (only one player allowed)	Limited but correct behavior	C
	Facebook, *social network*	T+O	Ct. profile Upd; repeat offline; "post" offline	Latest changes saved; offline disallowed; saved for retry	LWW for profile edits when online	C
	Instagram, *social network*	T+O	Ct. profile Upd online; repeat offline; add/Del comments to "posts"	Latest profile saved; offline ops fail; comments re-ordered through server timestamps	LWW → correct behavior	C
	Pandora, *music streaming*	T+O	Ct. Upd of radio station online; repeat offline; Ct. Del/Upd of radio station	Last update saved; offline ops fail; Upd fails but radio plays w/o notification of Del	Partial sync w/o, full sync w/ refresh. Confusing user semantics	S+E
	Pinterest, *social network*	T+O	Ct. "pinboard" creation and rename	Offline disallowed; sync on user refresh	LWW → correct behavior	E
	TomDroid⋆, *shared notes*	T	Ct. Upd; Ct. Del/Upd	Requires user refresh before Upd, assumes single writer on latest state; data loss	Incorrect assumption; **LWW → clobber**	E
	Tumblr, *blogging*	T+O	Ct. Upd of posts both online; repeat offline; Ct.Del/Upd	Later note posted; saved for retry upon online; app crash and/or forced user logout	**LWW → clobber; further bad semantics**	E
	Twitter, *social network*	T+O	Ct. profile Upd; online tweet; offline tweet	Stale profile on user refresh, re-sync to last save only on app restart; tweet is appended; offline tweets fail, saved as draft	LWW → correct behavior. Limited offline & edit functionality needed	C
	YouTube, *video streaming*	T+O	Ct. edit of video title online; repeat offline	Last change saved, title refresh on play; offline disallowed. Sync on user refresh	LWW → correct behavior	E
Also act as Sync Services	Box, *cloud storage*	T+O	Ct. Upd; Ct. Del/Upd; repeat offline	Last update saved; on Del, permission denied error for Upd; offline read-only	**LWW → clobber**	C
	Dropbox, *cloud storage*	T+O	Ct. Upd; Ct. Del/Upd (rename)	Conflict detected, saved as separate file; first op succeeds, second fails and forces refresh	Changes allowed only on latest version; FWW	C
	Evernote, *shared notes*	T+O	Ct. note Upd; Ct. Del/Upd; repeat offline; "rich" note sync failure	Conflict detected, separate note saved; offline handled correctly; partial/inconsistent note visible	Correct multi-writer behavior; **atomicity violation under sync**	C
	Google Drive, *cloud storage*	T+O	Ct. Upd (rename); Ct. Del/Upd	Last rename applies; file deleted first and edit discarded, or delete applied last and edit lost	**LWW → clobber**	C
	Google Docs, *cloud storage*	T+O	Ct. edit online; repeat offline	Real-time sync of edits; offline edits disallowed, limited read-only	Track & exchange cursor position frequently	S

Table 1: **Study of mobile app consistency.** Conducted for apps that use sync platforms, roll their own platform, and are platforms themselves. For brevity, only an *interesting* subset of tests are described per app. DM:data model, T:tables, O:objects, Ct.:concurrent, Del:delete, Upd:update; FWW/LWW: first/last-writer wins. CS:consistency scheme, S:strong, C:causal, E:eventual. ⋆:open-source. Inconsistent behavior is in bold. ";" delimited reasoning corresponds to separate tests.

and *eventual*. We place apps which violate both strong and causal consistency into the eventual bin, those which violate only strong consistency into the causal bin, and those which do not violate strong consistency into the strong bin. From among the many possible consistency semantics, we chose these three broad bins to keep the tests relatively simple.

2.3 Findings

Table 1 lists the apps and our findings; we make the following observations.

- **Diverse consistency needs**. In some cases, the same app used a different level of consistency for different types of data. For example, while Evernote provides causal consistency for syncing notes, in-app purchases of additional storage use strong consistency. Eventual and causal consistency were more popular while strong consistency was limited to a few apps (*e.g.*, Google Docs, Amazon). Many apps (*e.g.*, Hipmunk, Hiyu, Fetchnotes) exhibited eventual consistency whereas ones that anticipated collaborative work (*e.g.*, Evernote, Dropbox, Box) used causal.
- **Sync semantics oblivious to consistency**. Many apps used sync platforms which provide *last-writer-wins* semantics; while perfectly reasonable for append-only and single-user apps, in the absence of a programmatic merge [6], it led to data clobbering, corruption, and resurrection of deleted data in apps that allow for concurrent updates (*e.g.*, Keepass2Android, Hiyu, Township).
- **Limited offline support**. Offline support was not seamless on some apps (*e.g.*, Fetchnotes hangs indefinitely at launch), while other apps disabled offline operations altogether (*e.g.*, Township).
- **Inadequate error propagation**. When faced with a potential inconsistency, instead of notifying the user and waiting for corrective action, apps exhibit a variety of ad-hoc behavior such as forced user-logouts and crashes (*e.g.*, Tumblr), loss of in-app purchases (*e.g.*, Township), and double-or-nothing grocery purchases (*e.g.*, Hiyu).
- **Atomicity violation of granular data**. Apps need to store inter-dependent structured and unstructured data but the existing notion of sync is either table- or file-only (see Table 2) leading to inconsistencies. We define an *atomicity violation* as the state where only a portion of inter-dependent data is accessible. For example, Evernote, a popular note-taking app, allows its users to create *rich* notes by embedding a text note with multi-media, and claims no "half-formed" notes or "dangling" pointers [16]; however, if the user gets disconnected during note sync, we observe that indeed half-formed notes, and notes with dangling pointers, are visible on the other client.

To summarize, while valuable user and app data resides in mobile apps, we found that the treatment of data consistency and granularity is inconsistent and inadequate.

App/Platform	Consistency	Table	Object	Table+Object
Parse	E	√	×	×
Kinvey	E	√	×	×
Google Docs	S	√	√	×
Evernote	S or C	√	√	×
iCloud	E	×	√	×
Dropbox	S or C	√	√	×
Simba	S or C or E	√	√	√

Table 2: **Comparison of data granularity and consistency.** Shows the consistency and granularity offered by existing systems.

2.4 Case-study: Keepass2Android

Overview: Keepass2Android is a password manager app which stores account credentials and provides sync with multiple services; we chose Dropbox. On two devices using the same Dropbox account, we perform concurrent updates in two scenarios: 1) both devices are online, and 2) one device is offline. Device 1 (2) modifies credentials for accounts A and B (B and C).

Findings: On conflicts, the app prompts the user whether to resolve the conflict via a *merge* or an *overwrite*.

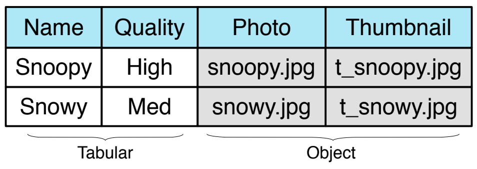
- *Scenario 1*: Changes are synced immediately and conflicts are resolved using a last-writer-wins strategy.
- *Scenario 2*: Device 2 makes offline changes. Upon syncing and choosing *merge*, its changes to account C are committed, but account B retains the change from Device 1. As a result, Device 2's changes to account B are silently lost. Also, the chosen conflict resolution strategy is applied for all offline changes, without further inspection by the user. This can result in inadvertent overwrites.

Thus, under concurrent updates, the conflict resolution strategy employed by this app results in an arbitrary merge or overwrite which leads to inconsistency. We describe how we fix an app with similar inconsistent behavior in §6.5.

3. Overview of Simba

Based on our study, we find that existing solutions for cloud-synchronized mobile data fall short on two accounts: inadequate data granularity (only tables or objects, but not both) and lack of *end-to-end* tunable consistency (an app must choose the same consistency semantics for all of its data). Table 2 compares the choice of data granularity and consistency offered by several popular data management systems.

To address these shortcomings, we develop Simba, a cloud-based data-sync service for mobile apps. Simba offers a novel data sync abstraction, sTable (short for Simba Table), which lets developers follow a convenient local-programming model; all app and user data stored in sTables seamlessly gets synchronized with the cloud and on to other devices. A sTable provides apps with end-to-end consistency over a data model unifying tables and objects. Each row of a sTable, called a sRow, can contain inter-dependent tabular and object data, and the developer can specify the distributed consistency for the table as a whole from among a set of options. *Simba thus treats a table as the unit of*



Name	Quality	Photo	Thumbnail
Snoopy	High	snoopy.jpg	t_snoopy.jpg
Snowy	Med	snowy.jpg	t_snowy.jpg

Tabular
Object

Figure 1: **sTable logical abstraction.**

	Strong$_S$	Causal$_S$	Eventual$_S$
Local writes allowed?	No	Yes	Yes
Local reads allowed?	Yes	Yes	Yes
Conflicts resolution necessary?	No	Yes	No

Table 3: **Summary of Simba's consistency schemes.**

consistency specification, and a row as the unit of atomicity preservation.

3.1 Choice of Data Granularity and Atomicity

Our study highlighted the need for treating inter-dependent coarse-grained data as the unit of atomicity under sync. A sTable's schema allows for columns with primitive data types (INT, BOOL, VARCHAR, etc) and columns with type *object* to be specified at table creation. All operations to tabular and object data stored in a sTable row are guaranteed to be atomic under local, sync, and cloud-store operations. Table-only and object-only data models are trivially supported. To our knowledge, our sTable abstraction is the first to provide atomicity guarantees across unified tabular and object rows. Figure 1 depicts the sTable logical layout containing one or more app-specified columns (physical layout in §4.1). In this example, the sTable is used by a photo-share app to store its album. Each sTable row stores an image entry with its "name", "quality", "photo" object, and "thumbnail" object.

3.2 Choice of Consistency

Mobile apps typically organize different classes of data into individual SQLite tables or files, grouping similar data, with the expectation that it is handled identically. Therefore, we choose to permit consistency specification per-table, as opposed to per-row or per-request, to be compatible with this practice. Thus, all tabular and object data in a sTable is subject to the same consistency. This enables an app to create different sTables for different kinds of data and independently specify their consistency. For example, a notes app can store user notes in one table and usage/crash data in another; the former may be chosen as causal or strong, while the latter as eventual.

Although sufficient for many apps, previous work [13] has shown that eventual consistency is not adequate for all scenarios. For example, financial transactions (*e.g.*, in-app purchases, banking) and real-time interaction (*e.g.*, collaborative document editing, reservation systems) require stronger consistency. Based on our study of the requirements of mobile apps (§2) and research surveys [31, 47, 54], sTable initially supports three commonly used consistency schemes; more can be added in the future. Our consistency schemes as described in Table 3 are similar to the ones in Pileus [55], and we borrow from their definitions. In all three schemes, reads always return locally stored data, and the primary differences are with respect to writes and conflict resolution.

- **Strong$_S$: All writes to a sTable row are serializable.** Writes are allowed only when connected, and local replicas are kept synchronously up to date. There are no conflicts in this model and reads are always local. When disconnected, writes are disabled, but reads, to potentially stale data, are still allowed. On reconnection, downstream sync is required before writes can occur. In contrast to strict consistency, which would not allow local replicas or offline reads, we provide sequential consistency [27] as a pragmatic trade-off based on the needs of several apps which require strong consistency for writes but allow disconnected reads, enabling both power and bandwidth savings. *Example:* Editing a document in Google Docs.
- **Causal$_S$: A write raises a conflict if and only if the client has not previously read the latest causally preceding write for that sTable row.** Causality is determined on a per-row basis; a causally-preceding write is defined as a write to the same row which is synced to the cloud before the current operation. Writes and reads are always local first, and changes are synced with the server in the background. Unlike Strong$_S$, causal consistency does not need to prevent conflicts under concurrent writers; instead, sTables provide mechanisms for automated and user-assisted conflict resolution. When disconnected, both reads and writes are allowed. *Example:* Syncing notes in Evernote.
- **Eventual$_S$: Last writer wins.** Reads and writes are allowed in all cases. Causality checking on the server is disabled for sTables using this scheme, resulting in last-writer-wins semantics under conflicts; consequently, app developers need not handle resolution. This model is often sufficient for many append-only and single writer scenarios, but can cause an inconsistency if used for concurrent writers. *Example:* "Starring" coupons on RetailMeNot.

3.3 Simba API

Designing the API that Simba exports to app developers requires us to decide on three issues: how apps set sTable properties, how apps access their data, and how Simba pushes new data to any app and enables the app to perform conflict resolution. Table 4 summarizes the API; any app written to this API is referred to as a Simba-app.

sTable creation and properties. The API allows for apps to set various properties on sTables. An app can set per-sTable properties either on table creation via *properties* in *createTable*, or through the various sync operations (*registerReadSync*, *registerWriteSync*, etc.) via *syncprefs*. For example, consistency is set on table creation, while sync-periodicity can be set, and reset, any time later.

CRUD (on tables and objects)
createTable(table, schema, properties)
updateTable(table, properties)
dropTable(table)
outputStream[] ← writeData(table, tblData, objColNames)
outputStream[] ← updateData(table, tblData, objNames, selection)
inputStream[] ← rowCursor ← readData(table, projection, selection)
deleteData(table, selection)
Table and Object Synchronization
registerWriteSync(table, period, delayTolerance, syncprefs)
unregisterWriteSync(table)
writeSyncNow(table)
registerReadSync(table, period, delayTolerance, syncprefs)
unregisterReadSync(table)
readSyncNow(table)
Upcalls
newDataAvailable(table, numRows)
dataConflict(table, numConflictRows)
Conflict Resolution
beginCR(table)
getConflictedRows(table)
resolveConflict(table, row, choice)
endCR(table)

Table 4: **Simba API for data management by mobile apps.**

Accessing tables and objects. sTables can be read and updated with SQL-like queries that can have a selection and projection clause, which enables bulk, multi-row local operations. In addition to the baseline CRUD operations, the API supports streaming read/write access to objects in order to preserve familiar file I/O semantics; objects are not directly addressable, instead streams to objects are returned through write (*writeData*, *updateData*) and read (*readData*) operations, respectively, on the enclosing row. Unlike relational databases, this makes it possible to support much larger objects as compared to SQL BLOBs (binary large objects) [36, 43], because the entire object need not be in memory during access.

Upcalls and conflict resolution. Every Simba-app registers two handlers: one for receiving an upcall on arrival of new data (*newDataAvailable*), and another for conflicted data (*dataConflict*). The latter upcall enables Simba to programmatically expose conflicts to apps (and onto users). Once notified of conflicted data, an app can call *beginCR* to explicitly enter the conflict resolution (CR) phase; therein, the app can obtain the list of conflicted rows in a sTable using *getConflictedRows* and resolve the conflicts using *resolveConflict*. For each row, the app can select either the client's version, the server's version, or specify altogether new data. When the app decides to exit the CR phase, after iterating over some or all of the conflicted rows, it calls *endCR*. Additional updates are disallowed during the CR phase. However, multiple updates on mobile clients can proceed normally during sync operations; if there is a conflict, Simba detects and presents it to the app. The conflict resolution mechanisms in Simba leave the decision of when to resolve conflicts up to individual apps, and enables them to continue making changes while conflicts are still pending.

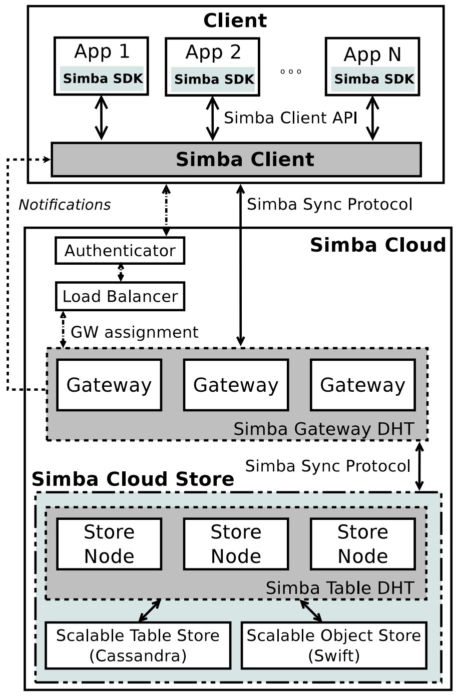


Figure 2: **Simba architecture.** Shows a sClient (runs as a background app on mobile devices) and the sCloud, a scalable cloud data-management system. Apps are written to the Simba API.

4. Simba Architecture and Design

We describe Simba's architecture and key design decisions that enable it to preserve atomicity and offer efficient sync.

4.1 Architecture

Simba consists of a client, sClient, and a server, sCloud, which communicate via a custom-built sync protocol. Figure 2 presents a high-level view of Simba's architecture.

Server. sCloud manages data across multiple sClients, sTables, and Simba-apps, and provides storage and sync of sTables with a choice of three different consistency schemes.

sCloud primarily consists of a data store, Simba Cloud Store (for short, Store), and client-facing Gateways. sClients authenticate via the authenticator and are assigned a Gateway by the load balancer. All subsequent communication between that sClient and the sCloud happens through the designated gateway. Store is responsible for storage and sync of client data, for both objects and tables; sTables are partitioned across Store nodes. The Gateway manages client connectivity and table subscriptions, sends notifications to clients, and routes sync data between sClients and Store. A gateway registers with all Store nodes for which it has client table subscriptions; it gets notified by the Store node on changes to a subscribed sTable. In the case of Strong_S consistency, update notifications are sent immediately to the client. For Causal_S and Eventual_S, notifications are sent periodically based on client-subscription periods.

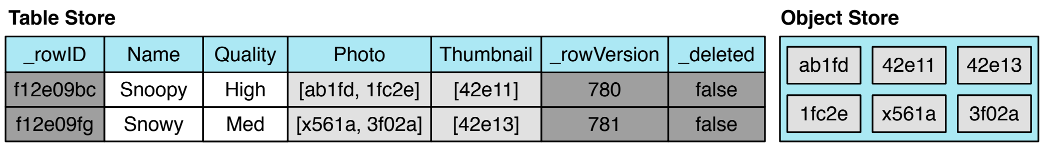


Table Store

_rowID	Name	Quality	Photo	Thumbnail	_rowVersion	_deleted
f12e09bc	Snoopy	High	[ab1fd, 1fc2e]	[42e11]	780	false
f12e09fg	Snowy	Med	[x561a, 3f02a]	[42e13]	781	false

Object Store

ab1fd	42e11	42e13
1fc2e	x561a	3f02a

Figure 3: **sTable physical layout.** Logical schema in Fig 1, light-shaded *object* columns contain chunk IDs, dark-shaded ones store Simba metadata.

Client $\rightleftharpoons$ Gateway

General

$\leftarrow$ *operationResponse(status, msg)*

Device Management

$\rightarrow$ *registerDevice(deviceID, userID, credentials)*
$\leftarrow$ *registerDeviceResponse(token)*

Table and Object Management

$\rightarrow$ *createTable(app, tbl, schema, consistency)*
$\rightarrow$ *dropTable(app, tbl)*

Subscription Management

$\rightarrow$ *subscribeTable(app, tbl, period, delayTolerance, version)*
$\leftarrow$ *subscribeResponse(schema, version)*
$\rightarrow$ *unsubscribeTable(app, tbl)*

Table and Object Synchronization

$\leftarrow$ *notify(bitmap)*
$\leftrightarrow$ *objectFragment(transID, oid, offset, data, EOF)*
$\rightarrow$ *pullRequest(app, tbl, currentVersion)*
$\leftarrow$ *pullResponse(app, tbl, dirtyRows, delRows, transID)*
$\rightarrow$ *syncRequest(app, tbl, dirtyRows, delRows, transID)*
$\leftarrow$ *syncResponse(app, tbl, result, syncedRows, conflictRows, transID)*
$\rightarrow$ *tornRowRequest(app, tbl, rowIDs)*
$\leftarrow$ *tornRowResponse(app, tbl, dirtyRows, delRows, transID)*

Gateway $\rightleftharpoons$ Store

Subscription Management

$\leftarrow$ *restoreClientSubscriptions(clientID, subs)*
$\rightarrow$ *saveClientSubscription(clientID, sub)*

Table Management

$\rightarrow$ *subscribeTable(app, tbl)*
$\leftarrow$ *tableVersionUpdateNotification(app, tbl, version)*

Table 5: **Simba Sync Protocol.** RPCs available between sClient and sCloud. Between sClient and sCloud, $\rightarrow$ / $\leftarrow$ signifies upstream / downstream. Between Gateway and Store, $\rightarrow$ / $\leftarrow$ signifies to / from Store.

sCloud needs to scale both with the number of clients and the amount of data (*i.e.*, sTables). To do so, we decouple the requirements by having independently scalable client management and data storage. sCloud is thus organized into separate distributed hash tables (DHTs) for Gateways and Store nodes. The former distributes clients across multiple gateways, whereas the latter distributes sTables across Store nodes such that each sTable is managed by at-most one Store node, for both its tabular and object data; this allows Store to serialize the sync operations on the same table *at the server*, offer atomicity over the unified view, and is sufficient for supporting all three types of consistency to the clients. Store keeps persistent sTable tabular and object data in two separate, scalable stores with the requirement that they support *read-my-writes* consistency [56].

Figure 3 shows the physical layout of a sTable for the photo-sharing app example in Figure 1. All tabular columns are mapped to equivalent columns in the table store; an *object* column (*e.g.*, Photo) is mapped to a column containing the list of its chunk IDs. The chunks are stored separately in the object store. A row in a table subscribed by multiple clients cannot be physically deleted until any conflicts get resolved; the "deleted" column stores this state.

Client. sClient is the client-side component of Simba which supports the CRUD-like Simba API. sClient provides reliable local-storage of data on the client, and data sync with sCloud, on behalf of all Simba-apps running on a device; the apps link with sClient through a lightweight library (Simba SDK). A detailed description of the client's fault-tolerant data store and network management is discussed elsewhere [20]. The paper describing the client focuses on client-side matters of failure handling and crash consistency; it shares the Simba API with this paper which in contrast delves into the Simba data-sync service and sCloud, focusing on end-to-end tunable distributed consistency.

Sync protocol. sCloud is designed to interact with clients for data sync and storage in contrast to traditional cloud-storage systems like Amazon S3 which are designed for storage alone; it thus communicates with sClient in terms of *change-sets* instead of *gets* and *puts*. Table 5 lists the messages that constitute Simba's sync protocol.[1] Rather than describe each sync message, for brevity, we present here the underlying high-level design rationale. For all three consistency models, any interested client needs to register a sync *intent* with the server in the form of a write and/or read subscription, separately for each table of interest.

sCloud is expected to provide multi-version concurrency control to gracefully support multiple independent writers; the sync protocol is thus designed in terms of versions. Simba's Strong_S and Causal_S consistency models require the ability to respectively avoid, and detect and resolve, conflicts; to do so efficiently in a weakly-connected environment, we develop a versioning scheme which uses compact version numbers instead of full version vectors [41]. Since all sClients sync to a centralized sCloud, Simba maintains a version number per row along with a unique row identifier. 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, allowing Simba to quickly identify which rows need to be synchronized. A similar scheme is used in gossip protocols [59]. The three consistency models also rely

[1] Limited public information is available on Dropbox's sync protocol. Drago *et al.* [17] deconstruct its data transfer using a network proxy; Dropbox uses 4MB chunks with delta encoding. We expect the high-level sync protocol of Dropbox to be similar to ours but do not know for certain.

on the versioning scheme to determine the list of sRows that have changed, or the change-set.

Internal to Simba, objects are stored and synced as a collection of fixed-size chunks for efficient network transfer (explained in §4.3). Chunking is transparent to the client API; apps continue to locally read/write objects as streams. Simba's sync protocol can also be extended in the future to support streaming access to large objects (*e.g.*, videos).

Downstream Sync (server to clients). Irrespective of the consistency model, first, the server notifies the client of new data; a $notify$ message is a boolean bitmap of the client's subscribed tables with only the modified sTables set. Second, the client sends a message ($pullRequest$) to the server with the latest local table version of the sTables. Third, the server then constructs a change-set with data from the client's table version to the server's current table version. Finally, the server sends the change-set to the client ($pullResponse$) with each new/updated chunk in an *objectFragment* message. The client applies the per-row changes to its local store and identifies conflicts. For Strong$_S$, the sync is immediate, whereas for Causal$_S$ and Eventual$_S$, the read subscription needs to specify a time *period* as the frequency with which the server notifies the client of changes.

Upstream Sync (clients to server). First, the client sends a *syncRequest* message with the change-set to the server, followed by an *objectFragment* message for every new/updated chunk. Second, the server absorbs the changes and sends a *syncResponse* message with a set of per-row successes or conflicts. For Strong$_S$, each client-local write results in a blocking upstream sync; for Causal$_S$ and Eventual$_S$, the client tracks dirty rows to be synced in the future. The sync change-set contains a list of sRows and their versions.

4.2 Ensuring Atomicity

End-to-end atomicity of unified rows is a key challenge in Simba. Updates across tabular and object data must preserve atomicity on the client, server, and under network sync; dangling pointers (i.e., table cells that refer to old or non-existent versions of object data) should never exist. Our design of Simba addresses this need for atomicity by using a specialized sync protocol and via a judicious co-design of the client and server.

Server. The Store is responsible for atomically persisting unified rows. Under normal operation (no failures), it is still important for Store to not persist half-formed rows, since doing so will violate consistency. For Strong$_S$, concurrent changes to the same sRow are serialized at the server. Only one client at a time is allowed to perform the upstream sync; the operation fails for all other clients, and the conflict must be resolved in sClient before retrying. Strong$_S$ also requires at-most a single row to be part of the change-set at a time. For Causal$_S$ and Eventual$_S$, upon receipt by the server, the change-set is processed row-by-row. Preserving atomicity is harder under failures as discussed next.

Store crash: Since Store nodes can crash anytime, Store needs to clean up orphaned chunks. Store maintains a *status log* to assess whether rows were updated in their entirety or not; each log entry consists of the row ID, version, tabular data, chunk IDs, and status (*i.e.*, *old* or *new*). Store first writes new chunks out-of-place to the object store when an object is created or updated. The sync protocol is designed to provide transaction markers for the server and client to determine when a row is ready for local persistence. Once all the data belonging to a unified row arrives at the Store, it atomically updates the row in the tabular store with the new chunk IDs. Subsequently, Store deletes the old chunks and marks the entry *new*. In the case of a crash that occurs after a row update begins but before it is completely applied, Store is responsible for reverting to a consistent state. An incomplete operation is rolled forward (*i.e.*, delete *old* chunks), if the tabular-store version matches the status-log version, or backwards (*i.e.*, delete *new* chunks), on a version mismatch. The status log enables garbage collection of orphaned objects without requiring the chunk data itself to be logged.

sClient crash: In the event of a client crash or disconnection midway through an upstream sync, sCloud needs to revert to a known consistent state. If a sync is disrupted, a gateway initiates an abort of the transaction on all destination Store nodes by sending a message; upon receipt of this message, each Store node performs crash recovery as explained above. To make forward progress, the client is designed to retry the sync transaction once it recovers (from either a crash or network disruption).

Gateway crash: Gateways can also crash; since they are responsible for all client-facing communication, their resiliency and fast recovery is crucial. To achieve this objective, Gateway is designed to only maintain *soft state* about clients which can be recovered through either the Store or the client itself; Gateways store in-memory state for all ongoing sync transactions. A failed gateway can be easily replaced with another gateway in its entirety, or its key space can be quickly shared with the entire gateway ring. As a result, gateway failures are quick to recover from, appearing as a short-lived network latency to clients. Gateway client-state is re-constructed as part of the client's subsequent connection handshake. Gateway subscriptions on Store nodes are maintained only in-memory and hence no recovery is needed; when a gateway loses connectivity to a Store node due to a network partition or a Store node crash, it re-subscribes the relevant tables on connection re-establishment.

Client. sClient plays a vital role in providing atomicity and supporting the different consistency levels. For Strong$_S$, sClient needs to confirm writes with the server before updating the local replica, whereas Causal$_S$ and Eventual$_S$ consistency result in the local replica being updated first, followed by background sync. Also, sClient needs to pull data from the server to ensure the local replica is kept up-to-date; for Strong$_S$, updates are fetched immediately as notifications are received, whereas Causal$_S$ and Eventual$_S$ may delay up to a configurable amount of time (*delay tolerance*).

Since the device can suffer network disruptions, and Simba-apps, the device, and sClient itself can crash, it needs to ensure atomicity of row operations under all device-local failures. Similar to sCloud, sClient performs all-or-nothing updates to rows (including any object columns) using a journal. Newly retrieved changes from the server are initially stored in a shadow table and then processed row-by-row; at this time, non-conflicting data is updated in the main table and conflicts are stored in a separate *conflict table* until explicitly resolved by the user. A detailed description of sClient fault-tolerance and conflict resolution is presented elsewhere [20]. Simba currently handles atomic transactions on individual rows; we leave atomic multi-row transactions for future work.

4.3 Efficient Sync

Simba needs to be frugal in consuming power and bandwidth for mobile clients and efficiently perform repeated sync operations on the server.

Versioning. The granularity of our versioning scheme is an important design consideration since it provides a trade-off between the size of data transferred over the network and the metadata overhead. At one extreme, coarse-grained versioning on an entire table is undesirable since it amplifies the granularity of data transfer. At the other extreme, a version for every table cell, or every chunk in the case of objects, allows fine-grained data transfer but results in high metadata overhead; such a scheme can be desirable for apps with real-time collaboration (*e.g.*, Google Docs) but is unsuitable for the majority. We chose a per-row version as it provides a practical middle ground for the common case.

Object chunking. Objects stored by apps can be arbitrarily large in practice. Since many mobile apps typically only make small modifications, to potentially medium or large-sized objects (*e.g.*, documents, video or photo editing, crash-log appends), re-sending entire objects over the network wastes bandwidth and power. Chunking is a common technique to reduce the amount of data exchanged over the network [34] and one we also use in Simba. In the case of a sRow with one or more objects, the sync change-set contains the *modified-only* chunks as identified through additional metadata; the chunks themselves are not versioned.

Change-set construction. Decoupling of data storage and client management makes the sCloud design scalable, but concurrent users of any one sTable are still constrained by update serialization on Store. Since data sync across multiple clients requires repeatedly ingesting changes and constructing change-sets, it needs to be done efficiently. The change-set construction requires the Store to issue queries on both row ID and version. Since the row ID is the default key for querying the tabular store, Store maintains an index on the version; this still does not address the problem that the version can help identify an entire row that has changed but not the objects' chunks within.

For upstream sync, sClient keeps track of dirty chunks; for downstream, we address the problem on the server through an in-memory *change cache* to keep track of per-chunk changes. As chunk changes are applied in the Store row-by-row, their key is inserted into the change cache.

5. Implementation

Client. sClient prototype is implemented on Android; however, the design is amenable to an iOS implementation as well. sClient is implemented as a background app which is accessed by Simba-apps via local RPC (AIDL [1] on Android); this design provides data services to multiple apps, allowing sClient to reduce network usage via data coalescing and compression. It maintains a single persistent TCP connection with the sCloud for both data and push notifications to prevent sub-optimal connection establishment and teardown [33] on behalf of the apps. sClient has a data store analogous to sCloud's Store; it uses LevelDB for objects and SQLite for table data.

Server. We implemented Store by using Cassandra [14] to store tabular data and Swift [39] for object data. To ensure high-availability, we configure Cassandra and Swift to use three-way replication; to support strong consistency, we appropriately specify the parameters (WriteConsistency=ALL, ReadConsistency=ONE). Since updates to existing objects in Swift only support eventual consistency, on an update, Store instead first creates a new object and later deletes the old one after the updated sRow is committed. Store assigns a read/write lock to each sTable ensuring exclusive write access for updates while preserving concurrent access to multiple threads for reading.

The change cache is implemented as a two-level map which allows lookups by both ID and version. This enables efficient lookups both during upstream sync, wherein a row needs to be looked up by its ID to determine if it exists on the server, and during downstream sync, to look up versions to determine change-sets. When constructing a change-set, the cache returns only the newest version of any chunk which has been changed. If a required version is not cached, the Store needs to query Cassandra. Since the Store cannot determine the subset of chunks that have changed, the entire row, including the objects, needs to be included in the change-set. Change-cache misses are thus quite expensive.

Sync protocol. It is built on the Netty [37] framework and uses zip data compression. Data is transmitted as Google Protobuf [7] messages over a TLS secure channel.

6. Evaluation

We seek to answer the following questions:

- How lightweight is the Simba Sync Protocol?
- How does sCloud perform on CRUD operations?
- How does sCloud scale with clients and tables?

Component	Total LOC
Gateway	2,145
Store	4,050
Shared libraries	3,243
Linux client	2,354

Table 6: **Lines of code for sCloud.** Counted using CLOC.

# of Rows	Object Size	Payload Size	Message Size (% Overhead)	Network Transfer Size (% Overhead)
1	None	1 B	101 B (99%)	133 B (99.2%)
	1 B	2 B	178 B (98.9%)	236 B (99.2%)
	64 KiB	64 KiB	64.17 KiB (0.3%)	64.34 KiB (0.5%)
100	None	100 B	2.41 KiB (96%)	694 B (85.6%)
	1 B	200 B	9.93 KiB (98%)	9.77 KiB (98%)
	64 KiB	6.25 MiB	6.26 MiB (0.2%)	6.27 MiB (0.3%)

Table 7: **Sync protocol overhead.** Cumulative sync protocol overhead for 1-row and 100-row *syncRequests* with varied payload sizes. Network overhead includes savings due to compression.

- How does consistency trade-off latency?
- How easy is it to write consistent apps with Simba?

sCloud is designed to service thousands of mobile clients, and thus, in order to evaluate the performance and scalability in §6.2 and §6.3, we create a Linux client. The client can spawn a configurable number of threads with either read or write subscriptions to a sTable, and issue I/O requests with configurable object and tabular data sizes. Each client thread can operate on unique or shared sets of rows. The chunk size for objects, the number of columns (and bytes per column) for rows, and consistency scheme are also configurable. The client also supports rate-limiting to mimic clients over 3G/4G/WiFi networks. This client makes it feasible to evaluate sCloud at scale without the need for a large-scale mobile device testbed. We run the client on server-class machines within the same rack as our sCloud deployment; these low-latency, powerful clients impose a more stringent workload than feasible with resource-constrained mobile devices, and thus, represent a worst-case usage scenario for sCloud. For evaluating latency trade-offs made by consistency choices in §6.4 and running our ported apps in §6.5, we interchangeably use Samsung Galaxy Nexus phones and an Asus Nexus 7 tablet, all running Android 4.2 and our sClient. In §6.5, we discuss how we use Simba to enable multi-consistency and to fix inconsistency in apps. We do not verify correctness under failures, as this is handled in [20]. sCloud consists of around 12K lines of Java code counted using CLOC [4] as shown in Table 6.

6.1 Sync Protocol Overhead

The primary objective of Simba is to provide a high-level abstraction for efficient sync of mobile application data. In doing so, we want to ensure that Simba does not add significant overhead to the sync process. Thus, we first demonstrate that Simba's sync protocol is lightweight. To do so, we measure sync overhead of rows with 1 byte of tabular data and (1) no, (2) a 1 byte, or (3) a 64 KiB object. We generate random bytes for the payload in an effort to reduce compressibility.

Operation	Processing time (ms)		
Upstream sync	Cassandra	Swift	Total
No object	7.3	-	26.0
64 KiB object, uncached	7.8	46.5	86.5
64 KiB object, cached	7.3	27.0	57.1
Downstream sync	Cassandra	Swift	Total
No object	5.8	-	16.7
64 KiB object, uncached	10.1	25.2	65.0
64 KiB object, cached	6.6	0.08	32.0

Table 8: **Server processing latency.** Median processing time in milliseconds; minimal load.

Table 7 quantifies the sync protocol overhead for a single message containing 1 row (*i.e.*, worst-case scenario), and a batch of 100 rows. For each scenario, we show the total size of the payload, message, and network transfer. We find that the baseline (no object data) message overhead for a single row with 1 byte of tabular data is 100 bytes. However, when batching 100 rows into a single sync message, the per-row baseline message overhead decreases by 76% to 24 bytes. Furthermore, as the payload (tabular or object) size increases, the message overhead quickly becomes negligible. Network overhead can be slightly higher in the single row cases due to encryption overhead; however, since the sync protocol is designed to benefit from coalescing and compression of data across multiple Simba-apps, the network overhead is reduced with batched rows. Overall, our results show that Simba Sync Protocol is indeed lightweight.

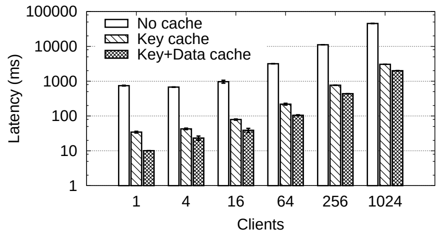
6.2 Performance of sCloud

In this section, we evaluate the upstream and downstream sync performance through a series of microbenchmarks. The intent of these benchmarks is simply to measure raw performance of the system under a variety of settings. We provision 48 machines from PRObE's Kodiak testbed [19]. Each node is equipped with dual AMD Opteron 2.6GHz CPUs, 8GB DRAM, two 7200RPM 1TB disks, and Gigabit Ethernet. The data plane switches are interconnected with 10 Gigabit Ethernet. We deploy an sCloud configuration with a single Store node and a single gateway. We configure Cassandra and Swift on disjoint 16-node clusters for backend storage; the remaining 16-nodes are used as client hosts. We use our Linux client to perform simple read or write operations on unique rows within a single sTable; Causal$_S$ consistency is used in all cases. Wherever applicable, we set the compressibility of object data to be 50% [24].

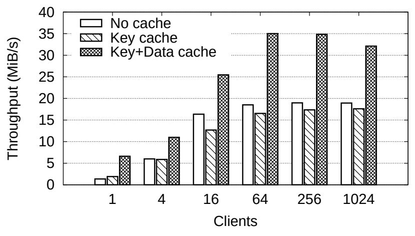
6.2.1 Downstream Sync

Our first microbenchmark measures the performance of downstream sync operations as experienced by clients, helping to understand the baseline performance while also highlighting the benefits of our change cache (described in §5).

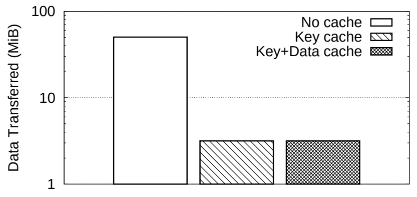
We run Store in three configurations: (1) No caching, (2) Change cache with row keys only, and (3) Change cache with row keys and chunk data. The chunk size is set to 64 KiB. For reference, the server-side sync processing time is in Table 8. In order to populate the Store with data, a writer



(a) Per-operation latency.

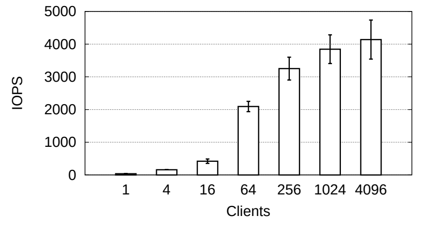


(b) Aggregate node throughput.

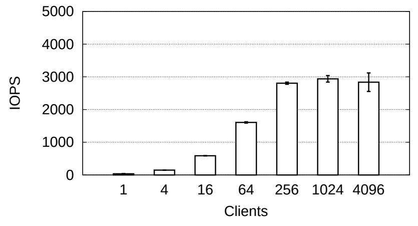


(c) Aggregate data transferred.

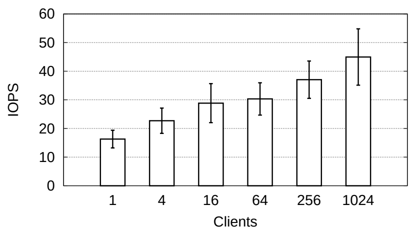
Figure 4: **Downstream sync performance for one Gateway and Store.** Note logscale on y-axis in (a) and (c).



(a) Gateway only.



(b) Store with table-only.



(c) Store with table+object.

Figure 5: **Upstream sync performance for one Gateway and Store.**

client inserts rows with 10 tabular columns totaling 1 KiB of tabular data, and 1 object column having 1 MiB objects; it then updates exactly 1 chunk per object. We then instantiate one or more reader clients, partitioned evenly over up to 16 nodes, to sync only the most recent change for each row.

Figure 4(a) shows the client-perceived latency as we vary the number of clients on the x-axis. As expected with no caching, clients experience the worst latency. When the change-cache stores keys only, the latency for 1024 clients reduces by a factor of 14.8; when caching both keys and data, latency reduces further by a factor of 1.53, for a cumulative factor of 22.8.

Figure 4(b) plots throughput on the y-axis in MiB/s. Firstly, with increasing number of clients, the aggregate throughput continues to increase up to 35 MiB/s for 256 clients. At this point, we reach the 64 KiB random read bandwidth of the disks. The throughput begins to decline for 1024 clients showing the scalability limits of a single Store node in this setting. Somewhat counter-intuitively, the throughput with the key cache appears to be less than with no cache at all; however, this is due to the very design of the Store. In the absence of a key cache, Store has to return the entire 1 MiB object, since it has no information of what chunks have changed, rather than only the single 64 KiB chunk which was updated. The throughput is higher simply because the Store sends entire objects; for the overall system, the important metrics are also client-latency, and the amount of data transferred over the network.

Figure 4(c) measures the data transferred over the network for a single client reading 100 rows. The graph shows that a no-cache system sends orders of magnitude more data compared to one which knows what chunks have changed. The amount of data transferred over the network is the same for the two caching strategies; the data cache only helps in reducing chunk data that needs to be fetched from Swift. Since Simba compresses data, the total data transferred over the network is less than the cumulative size.

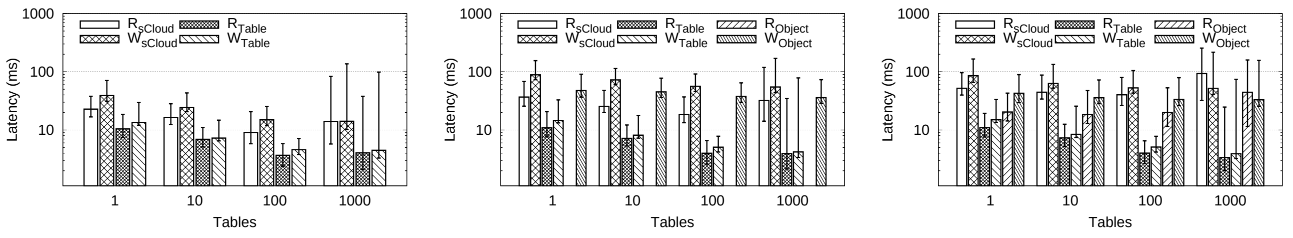
6.2.2 Upstream Sync

Next, we evaluate the performance of upstream sync. We run multiple writer clients partitioned evenly over up to 16 nodes. Each client performs 100 write operations, with a delay of 20 ms between each write, to simulate wireless WAN latency. We measure the total operations/second serviced for a varying number of clients. Note that due to the sync protocol, a single operation may require more than one message.

Figure 5 presents the client-perceived upstream performance. The first test stresses the Gateway alone by sending small control messages which the Gateway directly replies so that Store is not the bottleneck (Figure 5(a)). The figure shows that Gateway scales well up to 4096 clients. In the second test, we wrote rows with 1 KiB tabular data and no objects. This exercises Store with Cassandra but without Swift. As shown in Figure 5(b), Store performance peaks at 1024 clients, after which Cassandra latency starts becoming the bottleneck. For the third test, we wrote rows with 1 KiB of tabular data and one 64 KiB object. This exercises Store with both Cassandra and Swift. In Figure 5(c), we observe that the rate of operations is much lower as compared to table-only. This is due to two reasons: the amount of data being written is two orders of magnitude higher, and Swift exhibits high latency for concurrent 64 KiB object writes. In this case, 4096 clients caused contention to prevent reaching steady-state performance.

6.3 Scalability of sCloud

We now demonstrate the ability of sCloud to be performant at scale when servicing a large number of clients and tables. We provision 32 nodes from the PRObE Susitna cluster [19]. Each node is equipped with four 16-core AMD



(a) Table only. (b) Table+object, chunk cache enabled. (c) Table+object, chunk cache disabled.

Figure 6: **sCloud performance when scaling tables.**

Throughput (KiB/s)						
Tables	Table only		Table+Object w/ Cache		Table+Object w/o Cache	
	up	down	up	down	up	down
1	48	247	439	3,614	439	3,872
10	81	430	1,694	15,830	1,693	16,622
100	93	514	1,888	18,545	1,921	20,862
1K	255	2,369	2,259	78,412	2,282	77,980

Table 9: **sCloud throughput at scale.**

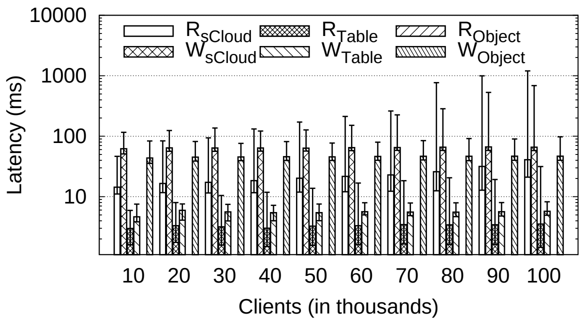


Figure 7: **sCloud performance when scaling clients.**

Opteron 2.1GHz CPUs, 128GB DRAM, two 3TB 7200RPM data disks, and an InfiniBand high-speed interface. We deploy an sCloud configuration with 16 Store nodes and 16 gateways. We configure Cassandra and Swift on disjoint 16-node clusters for backend storage; Cassandra is co-located with the gateways and Swift with the Store nodes.

We scale the number of tables and clients, and use the Linux client to instantiate multiple clients partitioned evenly across up to 16 nodes. Each client has either a read or a write subscription to a particular table. We set the ratio of read-to-write subscriptions as 9:1 and partition them evenly across tables. We vary the rate of requests per client to keep the aggregate rate of operations at 500/s across all scenarios.

6.3.1 Table Scalability

We examine three Store configurations: "table only", where each writer syncs rows with 1 KiB of tabular data, and "table+object" with and without the chunk data cache enabled, where each row also includes one 64 KiB object. We set the number of clients as $10x$ the number of tables.

Figure 6 measures latency on the y-axis, while the total number of tables (and clients) scales along the x-axis. The height of each bar depicts the median latency; error bars represent the 5th and 95th percentiles. Each set of bars measures client-perceived latency to *sCloud* and Store-perceived latency of the backend *Table* and *Object* stores for reads (R) and writes (W); this allows comparison of Cassandra's and Swift's contribution to the overall latency.

In the "table only" case (Figure 6(a)), we observe that client-perceived median latency for reads and writes decreases as the number of tables increases due to improved load distribution across the Store nodes. For the "table+object" cases, both with (Figure 6(b)) and without (Figure 6(c)) the chunk cache, we observe a similar decreasing trend in the Cassandra latency, and to a lesser extent, in the Swift write latency, as we scale. This is again due to better load distribution across the Store nodes. Without caching, Swift read latency remains stable until the 1000 table case, where load causes it to spike. We also observe that enabling the chunk data cache reduces the read latency as all chunks are served from memory, and slightly increases the write latency due to an increase in concurrent writes, as expected. In all cases, Cassandra tail-latency spikes in the 1000 table case, increasing the overall read and write latency. Although the user-perceived sCloud latency sharply rises for 1000 tables, we observe correlated latency spikes for Cassandra and Swift, suggesting the backend storage to be the culprit.

Table 9 shows the aggregate peak throughput of sCloud for each scenario. Throughput is lowest in the 1 table cases because performance is limited to that of a single Store node. For 10 and 100 tables, throughput is similar since the system is under-capacity, the number of operations/second is constant, and load distribution is not equal across all Store nodes. Throughput increases in the 1000 table case since more data is being transferred and better load distribution across Store nodes is achieved, which results in more efficient utilization of the available capacity.

Overall, sCloud scales well. Cassandra performance does degrade with large number of tables and thus can be substituted by a different table store in future versions of sCloud.

6.3.2 Client Scalability

In the previous case, we found that scaling the number of tables in Cassandra leads to significant performance overhead. Thus, we also investigate sCloud's ability to scale clients with fewer tables. Figure 7 shows the per-operation latency of sCloud on the y-axis while scaling from $10K$ to $100K$ clients along the x-axis, with the number of tables fixed at 128. In all cases, the median latency for all operations is less

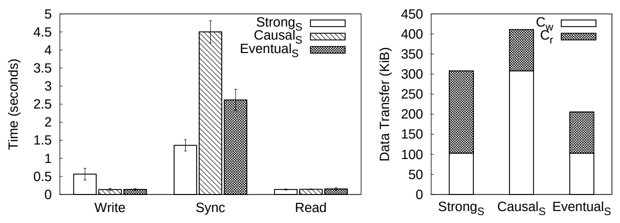


Figure 8: **Consistency comparison.** End-to-end latency and data transfer for each consistency scheme.

than 100 ms. Tail latency increases as we scale which we attribute to increased CPU load. Overall, we find that sCloud scales efficiently under high client load.

6.4 Trade-off: Consistency vs. Performance

On two Samsung Galaxy S3 phones running Android 4.2, we install sClient and a custom Simba-app which is able to read and write rows to a shared sTable. The devices connect over WPA-secured WiFi or a simulated 3G connection [52] using dummynet [18] to sCloud.

We perform end-to-end experiments with two different mobile clients, C_w (writer) and C_r (reader) and measure the Simba-app perceived latency for reads, writes, and data sync with the sCloud, under each of our consistency schemes for both 3G and WiFi simulated networks. The write payload is a single row with 20 bytes of text and one 100 KiB object. To differentiate between the consistency schemes, we use a third client C_c to write a row for the same row-key as C_w, which always occurs prior to C_w's write. We use a subscription period of 1 second for Causal$_S$ and Eventual$_S$ and ensure all updates occur before this period is over. Only C_r has a read subscription to the table.

Figure 8 shows the WiFi latency and associated data transfer (3G results similar; not shown). We show three latency values: 1) app perceived latency of update at C_w, shown as "Write", 2) sync-update latency from C_w to C_r, shown as "Sync", and 3) app-perceived latency for reading updated data at C_r, shown as "Read". We also plot the total data transferred by C_w and C_r for each consistency scheme.

Strong$_S$ has lowest sync latency as data is synced immediately, however, the client incurs network latency for the write operation, whereas writes are local in case of Causal$_S$ and Eventual$_S$. Immediate syncing in Strong$_S$ also causes higher data transfer because all updates must propagate immediately (*e.g.*, C_r must read both updates), not benefiting from overwrites or the change cache. Sync latency for Causal$_S$ is higher than Eventual$_S$ because the former requires more RTTs to resolve conflicts. With Causal$_S$, data transfer is inflated because the initial sync attempt by C_w fails, so C_w must read C_c's conflicting data, and retry its update after performing conflict resolution. Eventual$_S$ has the lowest data transfer due to last writer wins semantics and because C_r reads only the latest version since it syncs only once the read period expires. Without conflicts, the sync latency and data transfer for Causal$_S$ and Eventual$_S$ are similar (not shown). Finally, read latency is comparable for all consistency schemes because reads are always local; even with Strong$_S$, the local replica is kept up-to-date and reads do not communicate with the server.

6.5 Writing Simba Apps

Writing a multi-consistent app: We used an existing app, Todo.txt [8], to qualitatively evaluate the effort of writing an app that benefits from multiple consistency models. Todo.txt uses Dropbox to maintain and sync two files containing active tasks and archived tasks. We modified the app to store its data in two sTables. Active tasks can be modified frequently, so they are maintained with Strong$_S$ consistency, which ensures quick and consistent sync. Archived tasks cannot be modified, so it is sufficient to use Eventual$_S$ consistency. Any change to the archived task list is not immediately reflected on another device, but this is not critical to the operation of the app. Modifying Todo.txt to use Simba simplified the sync logic by eliminating the need for user-triggered sync, and allowed the app to use appropriate consistency models.

Fixing an inconsistent app: An open-source app in our study, Universal Password Manager (UPM), uses Dropbox to sync an encrypted database for user accounts. The app shows inconsistency when account information is changed concurrently on two devices and the database is synced with Dropbox; changes performed on one device get silently overwritten. To fix this inconsistency, we ported UPM to use Simba. We tried two approaches:

- Store the entire account database as an object in a sTable. This required fewer modifications; we simply used Simba instead of Dropbox to sync the database. However, conflict resolution is complex because conflicts occur at full-database granularity, so resolution needs to compare individual account information.
- Store each account as a separate row in a sTable; UPM no longer needs to implement its own database which eliminates the necessary logic for serialization and parsing the database file. Conflict resolution is made relatively simple because conflicts occur on a per-account granularity and can be easily handled.

In both approaches, Simba enables automated background sync based on one-time configuration, so the app's user doesn't need to explicitly trigger data sync with the cloud. In terms of developer effort, it took one of the co-authors of this paper a total of a few hours (< 5) to port UPM to Simba through both the above approaches.

7. Related Work

Geo-replication. Recently, several systems have examined various points in the trade-off between consistency, availability, and performance in the context of geo-distributed services. On the one hand, some systems (e.g., COPS [30] and Eiger [32]) have focused on providing low-latency causal consistency at scale, and others (e.g., Walter [50], Transaction Chains [63], and Red Blue consistency [29]) aim to minimize the latency incurred when offering other forms of *stronger-than-eventual* consistency, including serializability under limited conditions. On the other hand, systems like SPANStore [60] and Pileus [55] offer greater control for applications to select appropriate consistency across data centers, to reduce operating cost or to meet SLAs. While these systems manage data replication *across* and *within* data centers, we demonstrate and tackle the challenges associated with consistent replication across mobile devices with limited connectivity and bandwidth limitations.

Weakly-connected clients. Many prior efforts have studied data management in settings where clients are intermittently connected either to servers or to peers [12, 22, 26, 35, 45, 56]; Terry [54] presents an excellent synthesis on the topic. Coda [26] was one of the earliest systems to highlight the challenges in handling disconnected operations. Bayou [56] provides an API for application-specific conflict resolution to support optimistic updates in a disconnected system. Each of these systems chooses to implement the strongest possible consistency model given the availability constraints within which it operates; in contrast, motivated by our app study, Simba provides tunable consistency. Moreover, while previous systems treat either files or tables as the unit of consistency, Simba extends the granularity to include both, providing a more generic and useful abstraction.

Several data management systems for weakly connected clients explicitly focus on efficiency [34, 38, 57, 58]. Embracing the diverse needs of apps, Odyssey [38] provides OS support for applications to adjust the fidelity of their data based on network and battery conditions. Similarly, Simba provides mobile apps with programmatic means to adjust data fidelity. Cedar [58] provides efficient mobile database access by detecting commonality between client and server query results. Unlike Cedar, Simba does not rewrite queries, but applies them directly on the client replica. LBFS [34], a network file system designed for low-bandwidth, avoids redundant transfer of files by chunking files and identifying inter-file similarities; Simba's sync protocol applies similar data reduction techniques to only transmit modified chunks.

Sync services. Mobius [15] is a sync service for tables and provides fork-sequential consistency [40] which guarantees that clients see writes in the same order, even though locally client views can diverge; clients can also occasionally read uncommitted data. Simba provides a stronger consistency guarantee; clients see writes in the same order and never see uncommitted data. Simba also achieves this for both tables and objects. Like Mobius, Simba is not intended for peer-to-peer usage and leverages it for an efficient version scheme.

Dropbox does not store tables and files together; instead it provides a separate API for tables. However, little information is publicly available on Dropbox's client or server architecture. Drago *et al.* [17] collect and analyze Dropbox usage data but do not fully deconstruct its sync protocol.

CouchDB [2] along with its client TouchDB [9], an eventually-consistent distributed key-value store, provide "document" sync; this is the equivalent of a database row and consists of (often verbose) JSON objects. CouchDB's API is key–value and does not support large objects.

Sapphire [62] is a recent distributed programming platform for mobile/cloud applications. Sapphire *unifies* data and code into a single "virtual" node which benefit from its transparent handling of distributed systems tasks such as code-offloading, caching, and fault-tolerance. While useful for app execution, Sapphire does not fulfill all the needs for data management. Recent work on Pebbles [51] has shown that apps in fact heavily rely on structured data to manage unstructured objects; Simba's emphasis is on persistence and data sync over *unified* tabular and object data. The benefit of a unified table and object interface has been previously explored [46, 49] in the context of local systems; sTables extend it to networked mobile apps.

Cloud types [13] and SwiftCloud [61] are programming models for shared cloud data. Like Simba, both allow for local data copies to be stored on clients and later synced with the cloud. Unlike Simba, they require the programmer to handle synchronization.

8. Conclusions

Quoting Butler Lampson: *"the purpose of abstractions is to conceal undesirable properties; desirable ones should not be hidden"* [28]; existing abstractions for mobile data management either conceal too little (leave everything to the developer) or too much (one-size-fits-all data sync). By studying several popular mobile apps, we found varied consistency requirements within and across apps, inconsistent treatment of data consistency leading to loss and corruption of user data, and the inadequacy of existing data-sync services for the needs of mobile apps. Motivated by these observations, we proposed sTable, a novel synchronized-table abstraction—with tunable consistency, a unified data model for table and object data, and sync atomicity under failures—for developing cloud-connected mobile apps. We have built Simba, a data-sync service that provides app developers with an easy-to-use yet powerful sTable-based API for mobile apps; we have written a number of new apps, and ported a few existing apps, to demonstrate the utility of its data abstraction. Simba has been released as open-source under the Apache License 2.0 and is available at `https://github.com/SimbaService/Simba`.

9. Acknowledgements

We thank the anonymous reviewers and our shepherd Serdar Tasiran for their valuable feedback on earlier drafts of this paper. This material is supported in part by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE). http://www.nmc-probe.org/

References

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

[2] Apache CouchDB. http://couchdb.apache.org.

[3] Box Sync App. http://box.com.

[4] CLOC: Count Lines of Code. http://cloc.sourceforge.net.

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

[6] On Distributed Consistency - Part 5 - Many Writer Eventual Consistency. http://blog.mongodb.org/post/520888030/on-distributed-consistency-part-5-many-writer.

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

[8] Todo.txt. http://todotxt.com.

[9] TouchDB. http://tinyurl.com/touchdb.

[10] N. Agrawal, A. Aranya, and C. Ungureanu. Mobile data sync in a blink. In *HotStorage*, 2013.

[11] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *SoCC*, 2013.

[12] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006.

[13] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP*, 2012.

[14] Apache Cassandra Database. http://cassandra.apache.org.

[15] 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*, 2012.

[16] D. Engberg. Evernote Techblog: WhySQL? http://blog.evernote.com/tech/2012/02/23/whysql/, 2012.

[17] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *IMC*, 2012.

[18] Dummynet. http://info.iet.unipi.it/~luigi/dummynet/.

[19] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login*, 2013.

[20] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu. Reliable, Consistent, and Efficient Data Sync for Mobile Apps. In *FAST*, 2015.

[21] J. Gray. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling of Database management Systems*, 1976.

[22] R. G. Guy, J. S. Heidemann, W.-K. Mak, T. W. Page Jr, G. J. Popek, D. Rothmeier, et al. Implementation of the Ficus Replicated File System. In *USENIX Summer*, 1990.

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

[24] 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.

[25] iCloud for Developers. developer.apple.com/icloud.

[26] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM ToCS*, 1992.

[27] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE TC*, 1979.

[28] B. W. Lampson. Hints for Computer System Design. In *SOSP*, 1983.

[29] C. Li, D. Porto, A. Clement, J. Gehrke, N. Pregui, R. Rodrigues, A. Wieder, P. Bhatotia, A. Post, R. Rodrigues, et al. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *OSDI*, 2012.

[30] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[31] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. A short primer on causal consistency. *USENIX ;login*, 2013.

[32] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[33] J. C. Mogul. The case for persistent-connection HTTP. In *SIGCOMM*, 1995.

[34] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *SOSP*, 2001.

[35] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, 2002.

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

[37] Netty project. http://netty.io.

[38] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP*, 1997.

[39] OpenStack Swift. http://swift.openstack.org.

[40] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *DISC*, 2006.

[41] 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. *IEEE ToSE*, 1983.

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

[43] PostgreSQL. The Oversized-Attribute Storage Technique. http://www.postgresql.org/docs/9.3/static/storage-toast.html.

[44] S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE TPDS*, 2003.

[45] 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*, 2009.

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

[47] Y. Saito and M. Shapiro. Optimistic replication. *ACM CSUR*, 2005.

[48] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM ToCS*, 1984.

[49] 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*, 2013.

[50] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[51] R. Spahn, J. Bell, M. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser. Pebbles: Fine-grained data management abstractions for modern operating systems. In *OSDI*, 2014.

[52] SPDY Performance on Mobile Networks. `https://developers.google.com/speed/articles/spdy-for-mobile`.

[53] R. Swaby. With Sync Solved, Dropbox Squares Off With Apples iCloud. `http://tinyurl.com/dropbox-cr`, 2011.

[54] D. B. Terry. *Replicated Data Management for Mobile Computing*. Morgan & Claypool Publishers, 2008.

[55] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[56] 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*, 1995.

[57] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating Portable and Distributed Storage. In *FAST*, 2004.

[58] N. Tolia, M. Satyanarayanan, and A. Wolbach. Improving mobile database access over wide-area networks without degrading consistency. In *MobiSys*, 2007.

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

[60] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *SOSP*, 2013.

[61] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguia. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. *CoRR*, 2013.

[62] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *OSDI*, 2014.

[63] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.