UNIVERSITY OF CALIFORNIA
RIVERSIDE

Enabling Flexibility in Distributed Storage Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Dorian Jean Perkins

December 2015

The Dissertation of Dorian Jean Perkins is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

I am very grateful to my advisor, Harsha, without whose enthusiastic guidance and mentorship, I would not have been able to complete this dissertation. I am immensely thankful that he provided me with the opportunity to complete my doctoral degree when, due to unfortunate events, I was considering leaving grad school with my master's degree. Harsha poured genuine effort into seeing all of his students succeed and set a high bar which motivated us all to produce quality research. I can only hope to have as impactful of a career as he will.

I would also like to acknowledge the other professors who served on my committee: Srikanth, Vagelis, and Eamonn, and past members Michalis and Mart. It was a pleasure working with them all, and I appreciate their helpful feedback, insight, and support.

I am fortunate to have shared my journey with a great group of labmates, including Curtis Yu, Masoud Akhoondi, Zhe Wu, and Michael Butkiewicz, with whom I collaborated and pulled "all-nighters" before deadlines. They were always available to be a sounding board for ideas, assist in debugging and problem solving, and to trade stories.

I am also thankful to have interned at NEC Research Labs. My mentors Nitin Agrawal and Akshat Aranya were excellent hosts and I am continuously impressed by their passion and drive. I have become a more knowledgeable and confident researcher through my experience collaborating with them. I wish them continued success in their new endeavors. I would also like to thank my NEC office mate, Younghwan Go, whom I collaborated with and has been a great friend since our internship.

Victor Hill and Sean Mahoney from CS Systems deserve a hefty praise for assisting me to build, deploy, and configure my group's research cluster, as well as, for all other related technical assistance over the years. They have both been invaluable assets and their arduous work is often under-appreciated.

Another huge thank you to Amy Ricks and Madie Heersink for always taking care of any administrative issues that arose, as well as, Alicia Serrano for facilitating the equipment orders for our group's research cluster.

Most importantly, I am grateful to my mom for her unconditional love and support, and for always encouraging me to do my best. I am appreciative to my grandpa and my great-aunt for always uplifting and motivating me, as well as, the rest of my family for all of their support.

Last but not least, I would like to thank Steve Suh, Pamela Bhattacharya, Jason Ott, Bilson Campana, Ali Mohammadkhan, Indrajeet Singh, Huy Hang, and my labmates for their friendship, random distractions, and overall making grad school a more enjoyable experience. I blame all of you that I did not graduate sooner!

Acknowledgement of Previously Published Material. The text of this dissertation, in part, is a reprint of the material as is appears in the Proceedings of the Tenth European Conference on Computer Systems, April 2015. The co-author Harsha V. Madhyastha listed in that publication partially directed and supervised the research which forms the basis for a portion of this dissertation. The other co-authors provided technical expertise or assisted in experimental evaluation of the work.

In loving memory of

my grandma, Marcia, and my dad, Carlton –

may you both rest in peace.

ABSTRACT OF THE DISSERTATION

Enabling Flexibility in Distributed Storage Systems

by

Dorian Jean Perkins

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2015
Dr. Harsha V. Madhyastha, Co-Chairperson
Dr. Srikanth V. Krishnamurthy, Co-Chairperson

Over the last decade there has been a proliferation of distributed storage systems. These systems handle various types of data (e.g., text, photos, documents, files) in a variety of settings (e.g., enterprise, cloud, mobile). Typically, each system's design is optimized towards a specific workload and setting. Yet, when designing new applications, developers often face the onerous task of choosing one of these existing solutions to store their workload's data. This choice can be difficult for two reasons: (1) existing systems may not offer all of the features required by the new application and the development effort to extend an existing system's implementation can be prohibitive, and (2) choosing a system design which cost-effectively meets a workload's performance goal(s) is non-trivial and an incorrect choice may significantly inflate the cost to achieve the desired performance.

In this dissertation I design, build, and evaluate two distributed storage systems which, via their flexible implementations, offer improved ease of use and reduced development overhead when implementing data-centric applications.

First, I present *Simba*, a cloud-based data synchronization service which reduces development complexity of mobile apps. It improves upon existing systems which offer disjoint table- or object-only syncing with eventual consistency. Simba is built around a novel data abstraction 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.

Second, I present *Lego*, a modular emulator of distributed object stores. Lego is a tool which aids in the selection of the system design that can most cost-effectively satisfy a workload's performance goals. It enables developers to construct, deploy, and evaluate the performance of candidate object store designs in the target setting. Lego's modular architecture enables rapid prototyping, allowing for any object store design to be instantiated as a collection of modules. Since different object store designs often share similarities, this feature enables reuse of code across system implementations, thus significantly reducing development effort.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed storage systems have become a ubiquitous means of data storage in any scalable application or service. Popular large-scale internet applications, such as Facebook [34] and Twitter [94], are backed by one or more of these systems to store and serve their data. Distributed storage systems typically replicate data across many nodes within or across datacenters to provide benefits desirable to these applications, such as efficiently serving large volumes of data, ensuring availability, preserving consistency, and providing fault tolerance. There are many different flavors of distributed storage systems, each tuned towards handling a different class of data (e.g., text, photos, documents, files) in a specific setting (e.g., enterprise, cloud, mobile). Notable examples include GFS [35] to store web crawls at Google, Dynamo [27] to store shopping carts at Amazon, and Haystack [18] to store photos at Facebook.

When designing new applications, developers often face the onerous task of choosing one of these existing distributed storage solutions to store their workload's data. This choice can be difficult for two reasons. First, existing systems may not offer all of the features required by the new application. This leads to a situation where a developer must either extend an existing system to meet the requirements of their application, or design their own custom storage solution. Developing a production-quality distributed storage system is however an arduous task that can take several man-years of work. Thus, the development effort to extend an existing system's implementation or to build a custom solution can be prohibitive in terms of both time and cost. Second, choosing a system design which cost-effectively meets a workload's performance goal(s) is non-trivial. When faced with a new workload, it is difficult to predict which storage system will best meet the performance goals of the application. Furthermore, an incorrect choice may significantly inflate the cost to achieve the desired performance. Therefore, it is critical that developers make a carefully informed choice when choosing a distributed storage solution when developing new applications.

In this dissertation, my goal is to introduce flexibility into the design of distributed storage systems in order to offer improved ease of use and reduced development overhead when implementing data-centric applications. I design, build, and evaluate two distributed storage systems which fall under the categories of data synchronization services and distributed object stores. First, I present *Simba*, a cloud-based data synchronization service for mobile apps which offers a unified tabular and object data model. Simba reduces the complexity of designing mobile apps by enabling developers to effortlessly sync data with the cloud and other mobile devices while benefiting from end-to-end data consistency. Second, I present *Lego*, a emulator of distributed object stores which, via its modular architecture, exploits similarities between object store implementations to enable reuse of code and offer a rapid prototyping environment. Lego reduces developer effort to design, deploy, and evaluate candidate object store designs.

1.1 Reasons to make distributed storage systems flexible

In order to motivate the need for designing flexible distributed storage systems, I provide some insight into issues concerning both mobile data synchronization and distributed object stores, and how they might be mitigated with more flexible system designs.

1.1.1 Mobile data sync services

Mobile apps are interesting in that they must deal with the unique constraints of the devices on which they operate. Mobile devices are weakly-connected clients with frequent disconnections and mobile apps must tolerate this environment to provide a desirable user experience. A popular way of handling disconnected operations is to use a data-sync paradigm in which data is stored locally on the device and replicated to the cloud asynchronously. However, it is challenging to manage data consistency in such a setting due to 1) the limited network bandwidth and intermittent connectivity common to mobile devices, and 2) many apps storing inter-dependent structured and unstructured data. 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.

Existing mobile data synchronization services generally offer table-only or object-only syncing. Recently, Dropbox [30] began offering tabular data syncing [31] in addition to object data syncing, however, each has its own separate API which performs synchronization independently. Many mobile apps deal with both tabular and object data; for example, consider a photo album application which stores album metadata and the photo objects themselves. Using existing solutions, an application developer must store the metadata in one system and the photos in another system, later syncing them independently. If either of these transactions fail, it could leave half-formed data on the server to be synced to other mobile clients. Alternatively, a more desirable approach might be a service which unifies both tabular and object data and provides atomic synchronization across this data as a single logical unit.

Reasoning about consistency is challenging even for knowledgeable developers. Many of the available services offer only eventual consistency, leaving it up to the developer to understand how this affects their application's workflow. When an app needs stronger-than-eventual consistency, developers must find some way of ensuring that themselves, such as through a lock-file which protects against concurrent access. Instead, it would be beneficial to developers to have a service that allows for useful consistency semantics in various scenarios, such as strong consistency for monetary transactions, or causal consistency for versioned data, in addition to commonly-available eventual consistency.

1.1.2 Distributed object stores

The preponderance of distributed object stores partly stems from the need to tailor an object store's design to the workload characteristics and hardware configurations in the environment where the system is to be deployed. Due to this dependence of object store design on workload and hardware characteristics, as new workloads emerge and hardware configurations evolve, the need for new object stores is likely to continue. In light of this, distributed object stores could benefit by introducing flexibility both into how they are implemented and evaluated.

1.1.2.1 Design and implementation

When surveying existing distributed object stores, a trend that becomes apparent is the often similar set of components which exist across object store implementations. For example, both GFS [35] and Haystack [18] use a central directory which stores a mapping for each object to the servers at which it is stored. However, many of these implementations are highly-optimized, often having workload assumptions baked into their designs, which make them difficult to extend. With such stark similarities in designs, it would have been beneficial to the developers of these systems to have been able to easily reuse portions of the code which are identical in operation. Though their target workload or setting may significantly differ, enabling such reuse might be interesting for the design and prototyping phases of distributed object store development.

1.1.2.2 Evaluating performance

Due to the complex nature of distributed storage systems, one cannot simply perform back-of-the-envelope calculations to estimate their performance. For example, based on the manner in which a design stores and retrieves data and metadata on hard disks, SSDs, and RAM, I can attempt to estimate the performance offered by any particular scale of deployment by accounting for workload characteristics (e.g., object size distribution and GET:PUT ratio) and micro-benchmarks of hardware performance (e.g., disk seek times and network bandwidth). However, such an estimate will ignore delays introduced due to queueing both in the network and at storage servers. Moreover, it is hard to capture all the intricacies of storage hardware using simple micro-benchmarks, e.g., there are significant variances across different units of the same type of hard disk drive [50], and SSD performance depends on its firmware, whose logic can be hard to model. Therefore, it is important to capture the *dynamic interactions* between the input workload and the target hardware, in order to accurately estimate performance.

Empirically evaluating performance of a workload across a variety of systems is often impractical. A developer must endure the overhead of deployment and configuration, as well as, integrating their application by writing a driver for each of these candidate systems before they are able to evaluate performance of their workload. An integrated solution which allows evaluation multiple system designs on the target hardware without the need to explicitly configure and deploy each candidate system could save developers time and enable them to make a better-informed decision.

1.2 Thesis and Contributions

In this dissertation, I am supporting the following thesis: *enabling flexibility in distributed object stores can reduce development effort when designing and implementing new applications*.

The content of this dissertation describes work completed in the development of two distributed storage systems, *Simba* and *Lego*, which, via their flexible implementations, offer improved ease of use and reduced development overhead when implementing data-centric applications.

1.2.1 Cloud-based data sync for mobile apps

In this work, I 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, I propose a novel data abstraction, called a *Simba Table*, 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. I 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.

1.2.2 Emulation of distributed object stores

To aid the selection of the distributed object store design that can most cost-effectively satisfy performance goals, I design and implement Lego, an object store emulator. Lego enables developers to instantiate various system designs on a cluster of servers, compare the performance offered by them, and pick the design best suited to the target deployment scenario.

To enable rapid prototyping of any system design, I construct Lego as a modular architecture, wherein one can instantiate any object store as a collection of modules that are oblivious to each others' implementation. Since different object store designs often share some similarities, Lego's modularity enables new system designs to be instantiated by reusing modules written for previously instantiated designs; new implementations for only one (or some) of Lego's components will be necessary.

In using Lego to instantiate 10 popular distributed object stores, I found that Lego reduces the total number of lines of code by over 80% compared to independent implementations of each system, while accurately capturing cost-performance tradeoffs across workloads and hardware configurations.

1.3 Organization

The remainder of this dissertation is organized as follows: Chapter 2 provides background on distributed storage systems and previous work in the area, Chapters 3 and 4 describe my systems Simba and Lego, respectively, and finally I conclude my dissertation in Chapter 5.

Chapter 2

Background and Prior Work

In this chapter, I will introduce the background knowledge for the distributed storage systems and the prior related work that are the basis of my dissertation. This dissertation focuses on two types of distributed storage systems: data synchronization services and distributed object stores. While both of these systems offer distributed data storage, they differ in the interface which they expose to the developer. The first class, data synchronization services, offer background or explicit synchronization of changes between a local data store and a remote server (i.e., cloud storage) and/or across other devices. The second class, distributed object stores, offer a simple PUT/GET (i.e., store and retrieve) interface to enable developers to perform reads and writes directly to a specific object.

In the first section, I will discuss the current state of mobile data synchronization platforms. I will overview the popular services currently available on the market and discuss previous academic efforts related to this space. Following this, I will discuss distributed object stores and prior efforts which focus on emulation and rapid prototyping.

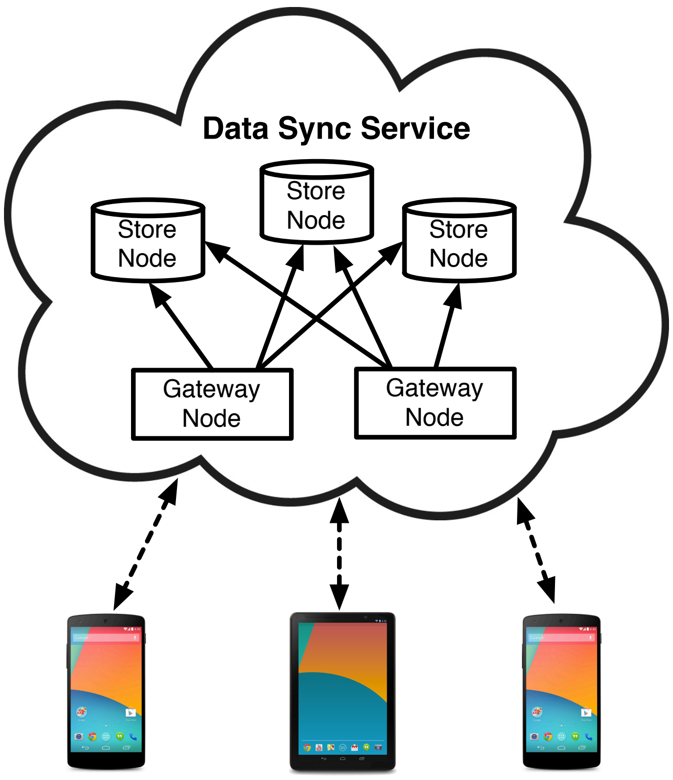


Figure 2.1: Example of a cloud-based data sync service.

2.1 Mobile data synchronization

Mobile devices provide unique challenges for developers when designing applications. Due to the nature of cellular networks, developers cannot assume devices will always be online, and thus, mobile apps must be able to tolerate frequent network disruptions as devices may often be disconnected from the carrier network. In response, mobile apps are often designed store all changes locally and await synchronization of these local modifications at a later time. Mobile apps commonly employ data synchronization services to enable synchronization of user data to and from the cloud and across other devices. Figure 2.2 depicts such a data synchronization service. Typically data synchronization is a background service which periodically polls for modified local or remote data, scheduling synchronization when new modifications are available (trivially supporting on-demand synchronization as well).

Most data-centric apps store inter-dependent unstructured (*e.g.*, photos and videos) and structured data (*e.g.*, album info and metadata), and this inter-dependent data needs to be managed not only locally, but also on cloud storage and across devices. Existing sync services offer either file-only (*e.g.*, Dropbox [86], iCloud [47], Box [4], and Google Drive [6]) or table-only (*e.g.*, Parse [73], Kinvey [48]) abstractions. This can lead to issues under failures, as atomicity across inter-dependent data can be violated, and consistency of data can be compromised if half-formed data is updated by a remote client.

While many apps chose weaker consistency semantics for the sake of availability and efficiency, if their data synchronization is designed poorly, this can lead to inconsistencies and loss of user data. Though several apps use existing commercial sync services (*e.g.*, Dropbox), even such apps exhibit undesirable behavior under concurrent use (see Table 3.1). The dominant reason being that apps choose the sync service oblivious to the ramifications of their last-writer-wins semantics on the data model. Additionally, I 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.

To improve the current state of mobile app development, I design and implement *Simba* to address these issues. Simba is a cloud-based data synchronization service which is built around a novel data abstraction, called a *Simba Table*, which unifies structured tabular and unstructured object data. Simba also offers tunable consistency at the granularity of a Simba Table, thus, developers can easily design apps which provide different consistency semantics depending on the data being stored.

2.1.1 Related work

There have been several academic and commercial efforts to which my research is related and some which I have borrowed ideas from. In this section, I discuss works that relate to building geo-replicated and data synchronization services, and handling weakly-connected clients.

2.1.1.1 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 [55] and Eiger [57]) have focused on providing low-latency causal consistency at scale, and others (e.g., Walter [83], Transaction Chains [106], and Red Blue consistency [54]) 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 [103] and Pileus [91] 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, I demonstrate and tackle the challenges associated with consistent replication across mobile devices with limited connectivity and bandwidth limitations.

2.1.1.2 Weakly-connected clients

Many prior efforts have studied data management in settings where clients are intermittently connected either to servers or to peers [49, 89, 78, 20, 65, 42]; Terry [90] presents an excellent synthesis on the topic. Coda [49] was one of the earliest systems to highlight the challenges in handling disconnected operations. Bayou [89] 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 my 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 [64, 92, 68, 93]. Embracing the diverse needs of apps, Odyssey [68] 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 [93] 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 [64], 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.

2.1.1.3 Sync services

Mobius [24] is a sync service for tables and provides fork-sequential consistency [71] 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.* [29] collect and analyze Dropbox usage data but do not fully deconstruct its sync protocol.

CouchDB [3] along with its client TouchDB [10], an eventually-consistent distributed key-value store, provide "document" sync; this is the equivalent of a database row and consists

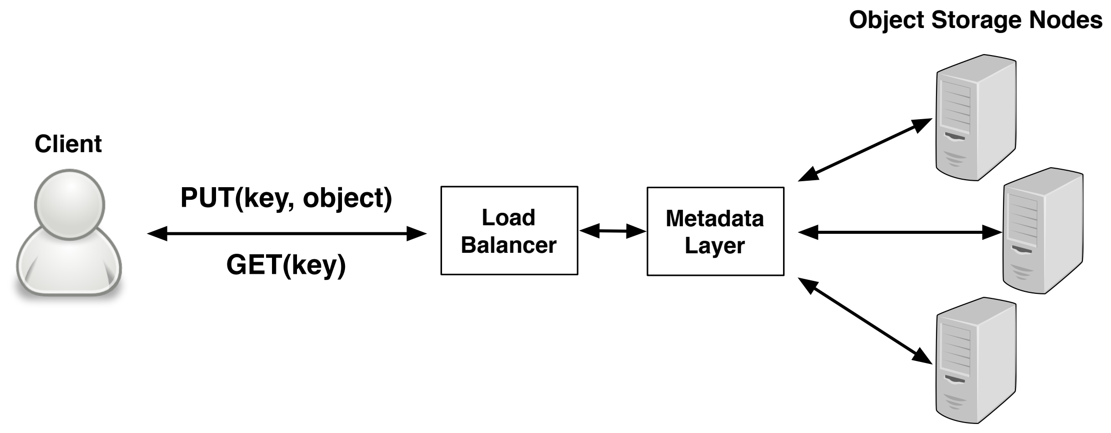


Figure 2.2: Example of a distributed object store.

of (often verbose) JSON objects. CouchDB's API is key–value and does not support large objects.

Sapphire [105] 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 [84] 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 [82, 79] in the context of local systems; sTables extend it to networked mobile apps.

Cloud types [22] and SwiftCloud [104] 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.

2.2 Distributed object stores

Object stores are a storage architecture that manages data as unstructured binary data, often called a BLOB (Binary Large OBject); herein, I will refer to this data as an *object*. An object is a piece of data, such as an image, audio, or other file; each object stored in an object store is typically mapped to a unique identifier. Object stores provide a layer of abstraction atop the lower level storage hardware exposing an interface that manages data as objects instead of files or disk blocks.

Distributed object stores span multiple physical servers and storage devices exporting an interface to the aggregate storage space such that it appears as one or more logical volumes. Figure [**?**] shows an overview of a distributed object store. Internally, distributed object stores spread data objects across the available hardware, handling aspects such as load balancing, data replication, data consistency, and fault tolerance. Metadata and data are typically stored separately for increased parallelism, as well as, to enable enforcement of various data management policies, such as centralized or distributed metadata directories. Metadata directories store the mappings between an objects key and its physical location(s). This metadata can be cached or indexed for quick lookup. The objects themselves are stored on storage devices in various arrangements. For example, many object stores choose a log-based data store, where new data objects are appended to the end of a log in order maintain sequential disk accesses and avoid disks seeks.

Distributed object stores have become a popular means of data storage in scalable applications and services which serve large amounts of object data. There is a wide breadth of distributed object stores available on the market, ranging from enterprise solutions to public-cloud offerings. Companies such as Amazon [1], Google [39], and Microsoft [11] all offer cloud-based object storage services which are available for any Internet-enabled application.

Ceph [101] and OpenStack Swift [87] are examples of open-source distributed object stores which can be deployed on a cluster of machines. Additionally, companies such as EMC also sell distributed object storage platforms for enterprise [17].

As popularity of distributed object stores has grown, so have the number of different systems available. Each of these systems is typically optimized for a specific workload and setting, and these assumptions can be baked into the design of the system. Not only is it difficult for developers to choose from the numerous available storage systems the system which best meets their workload's performance goals, but these optimizations often make it difficult to extend these systems as well. Thus, as discussed in Chapter 4, I design and implement Lego, a distributed object store emulator, in order to aid developers in choosing the system design that is best suited to a new deployment setting.

2.2.1 Related work

In this section, I discuss prior work which is relevant to building a modular emulator for distributed object stores, including configurable storage systems, emulation and simulation of complex systems, and rapid prototyping environments.

2.2.1.1 Configurable storage

Some prior efforts share our goal of a generic architecture for storage systems. PADS [19] is configurable to build a range of replication systems with varying consistency semantics, while Ursa Minor [12] varies erasure coding parameters to meet availability requirements. Boxwood [58] provides abstractions for distributed storage with fault-tolerance as the objective, whereas Anvil [60] focuses on enabling modularity in database systems. In contrast to all of these efforts, we focus on evaluating performance goals, rather than consistency, availability, or fault-tolerance.

2.2.1.2 Emulation and simulation

There has been considerable work on systems emulation/simulation frameworks. Network emulators [102, 96] popularized large-scale network emulation to enable reproducible experimentation of distributed systems. While these testbeds complement our emulation framework as a potential deployment environment, Lego focuses on distributed object store emulation. KVZone [38] provides workload generation to benchmark existing key-value stores, but requires development of an adapter for each new system it evaluates. Instead, Lego can emulate multiple storage configurations on the same platform. Exalt [100] and David [14] use compression to emulate large-scale deployments, yet these systems focus on scalability rather than performance.

Memulator [41] enables emulation of non-existent storage hardware using in-memory simulation. ns-3 [69] and DiskSim [28] provide simulation environments for network topologies and disk architectures, respectively. However, Lego focuses on emulation since it offers higher fidelity.

2.2.1.3 Rapid prototyping

Other work has targeted accelerated development using rapid prototyping environments. Neko [95] is a suite for developing distributed algorithms. Mininet [53] is a network emulator which enables self-contained SDN prototypes using OS-level virtualization. Click [62] introduced a modular software architecture for rapid prototyping of router architecture. Lego shares the goal of providing a rapid prototyping platform while focusing on distributed object stores.

Chapter 3

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

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 [55, 57, 54, 91] in recent times, application developers continue to struggle with writing applications that are correct and consistent [15].

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 chapter, I 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 [55, 57, 54, 91], 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 [49, 89, 65], 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 [13, 84], 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, I studied a number of popular mobile apps. I make four primary observations from my study. First, I 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 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 [86, 47, 4, 6] or table-only [73, 24, 43] abstractions. As a result, atomicity of granular data is not preserved under sync even for a popular app such as Evernote which claims so [26].

Mobile apps are ultimately responsible for the user experience and need to guarantee data consistency in an end-to-end manner [81]. Motivated by the observations from my study, I propose a novel data synchronization abstraction that offers powerful semantics for mobile apps. My proposed abstraction, which I 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 my knowledge, my 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, I 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.

I 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) my prototype scales with the number of users and the amount of data stored, as measured using the PRObE Kodiak and Susitna testbeds [36]. I 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.

3.1 Study of Mobile App Consistency

I 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. I adopted this terminology from Gray *et al.* [40], who present an excellent discussion on the relationship between consistency and granularity.

I chose a variety of apps for my 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 my study also have diverse functionality ranging from games, photo sharing, and social networking, to document editing, password management, and commerce.

	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

Continued on next page

	App, *Function* & Platform	**DM**	**Tests**	**User-visible Outcome**	**Reasoning**	**CS**
	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 3.1: 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. $\star$: open-source. Inconsistent behavior is in bold. ";" delimited reasoning corresponds to separate tests.

3.1.1 Methodology

I 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). I manually performed a series of operations on each app and made a qualitative assessment.

For consistency, my 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, I did not have access to the service internals; instead, I relied on user-initiated actions and user-visible observations on mobile devices.

For granularity, I 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 my hypothesis, I traced the local I/O activity to correlate accesses to SQLite and the file system.

First, for each app, I 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, I performed operations while one or both devices were offline. I then reconnected the devices and observed how offline operations propagated. Finally, I 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.

3.1.2 Classification

Our assessments are experimental and not meant as a proof; since automatic verification of consistency can be undecidable for an arbitrary program [77], my findings are evidence of (in)consistent behavior. I chose to classify the apps into three bins based on the test observations: *strong*, *causal*, and *eventual*. I 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, I chose these three broad bins to keep the tests relatively simple.

3.1.3 Findings

Table 3.1 lists the apps and my findings; I 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 [7], 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 3.2) leading to inconsistencies. I 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 [26]; however, if the user gets disconnected during note sync, I 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, I found that the treatment of data consistency and granularity is inconsistent and inadequate.

3.1.4 Case-study: Keepass2Android

Overview: Keepass2Android is a password manager app which stores account credentials and provides sync with multiple services; I chose Dropbox. On two devices using the same Dropbox account, I 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.

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	√	√	√

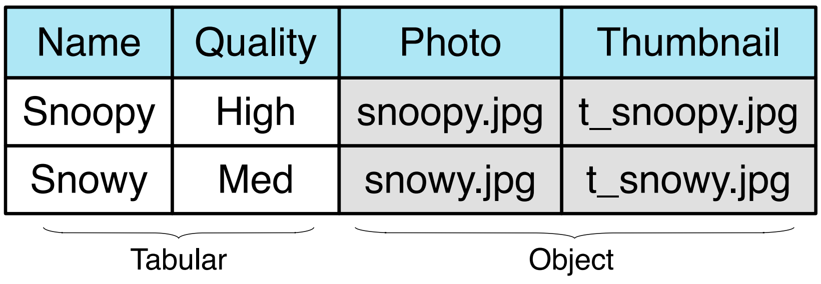
Table 3.2: Comparison of data granularity and consistency. Shows the consistency and granularity offered by existing systems.

Thus, under concurrent updates, the conflict resolution strategy employed by this app results in an arbitrary merge or overwrite which leads to inconsistency. I describe how I fix an app with similar inconsistent behavior in §3.5.5.

3.2 Overview of Simba

Based on my study, I 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 3.2 compares the choice of data granularity and consistency offered by several popular data management systems.

To address these shortcomings, I 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



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

Tabular
Object

Figure 3.1: sTable logical abstraction.

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

3.2.1 Choice of Data Granularity and Atomicity

My 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 my knowledge, my sTable abstraction is the first to provide atomicity guarantees across unified tabular and object rows. Figure 3.1 depicts the sTable logical layout containing one or more app-specified columns (physical layout in §3.3.1). 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.

	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 3.3: Summary of Simba's consistency schemes

3.2.2 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, I 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 [22] 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 my study of the requirements of mobile apps (§3.1) and research surveys [56, 90, 80], sTable initially supports three commonly used consistency schemes; more can be added in the future. My consistency schemes as described in Table 3.3 are similar to the ones in Pileus [91], and I 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, I provide sequential consistency [51] 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.

3.2.3 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

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 3.4: Simba API for data management by mobile apps.

data to any app and enables the app to perform conflict resolution. Table 3.4 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 *syncpref*s. For example, consistency is set on table creation, while sync-periodicity can be set, and reset, any time later.

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) [66, 75], 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.

3.3 Simba Architecture and Design

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

3.3.1 Architecture

Simba consists of a client, sClient, and a server, sCloud, which communicate via a custom-built sync protocol. Figure 3.2 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$_S$

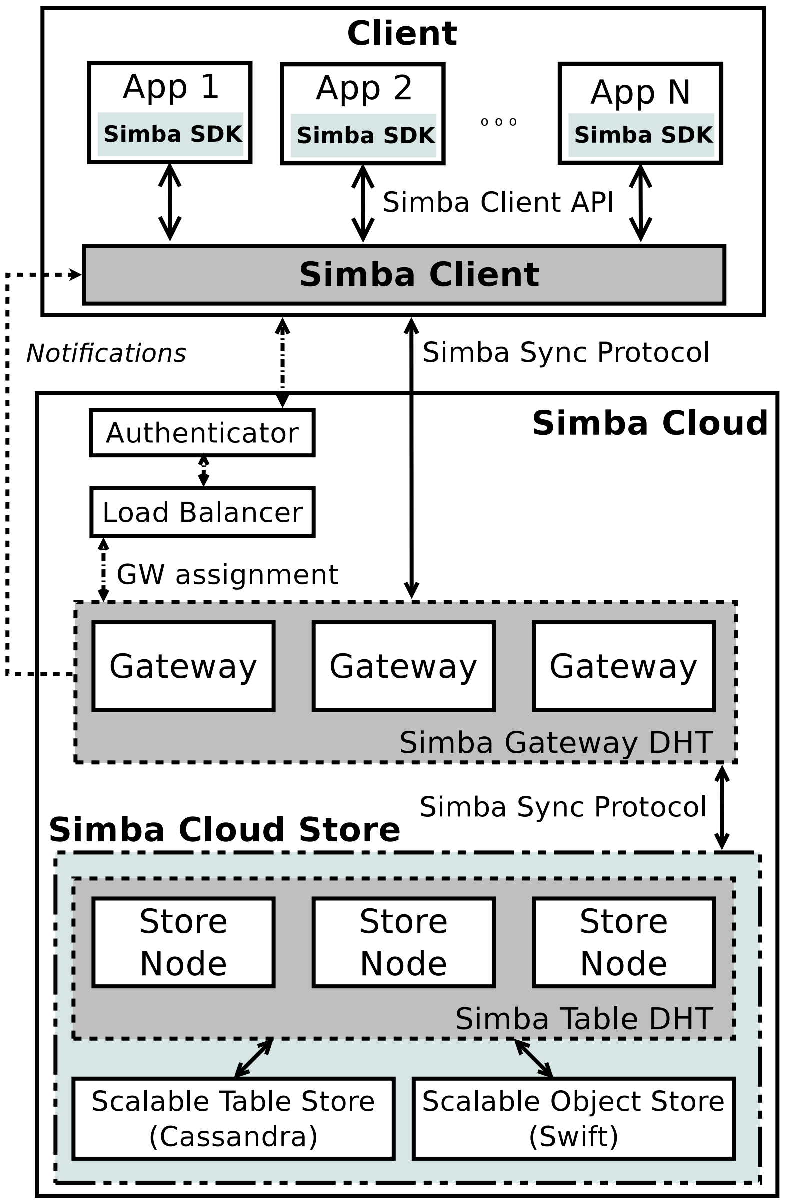


Figure 3.2: Simba architecture. Shows a sClient (runs as a background app on mobile devices) and the sCloud, a scalable cloud data-management system. Apps are written to the Simba API.

and Eventual$_S$, notifications are sent periodically based on client-subscription periods.

sCloud needs to scale both with the number of clients and the amount of data (*i.e.*, sTables). To do so, I 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 [89].

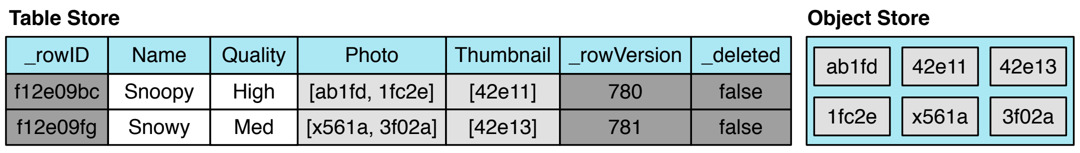


Table Store

_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 3.3: sTable physical layout. Logical schema in Fig 3.1, light-shaded *object* columns contain chunk IDs, dark-shaded ones store Simba metadata.

Figure 3.3 shows the physical layout of a sTable for the photo-sharing app example in Figure 3.1. 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 [37]. The work describing the client focuses on client-side matters of failure handling and crash consistency; it shares the Simba API with this work which in contrast delves into the Simba data-sync service and sCloud, focusing on end-to-end tunable distributed consistency.

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 3.5: 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.

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 3.5 lists the messages that constitute Simba's sync protocol.[1] Rather than describe each sync message, for brevity, I 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, I develop a versioning scheme which uses compact version numbers instead of full version vectors [72]. 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 [97]. The three consistency models also rely 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 §3.3.3). 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

[1]Limited public information is available on Dropbox's sync protocol. Drago *et al.* [29] deconstruct its data transfer using a network proxy; Dropbox uses 4MB chunks with delta encoding. I expect the high-level sync protocol of Dropbox to be similar to mine but do not know for certain.

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.

3.3.2 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. My 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 [37]. Simba currently handles atomic transactions on individual rows; I leave atomic multi-row transactions for future work.

3.3.3 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 my 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. I 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 [64] and one I 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, I 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.

3.4 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 [2] 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 [61] 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. I implemented Store by using Cassandra [23] to store tabular data and Swift [70] for object data. To ensure high-availability, I configure Cassandra and Swift to use three-way replication; to support strong consistency, I 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 [67] framework and uses zip data compression. Data is transmitted as Google Protobuf [8] messages over a TLS secure channel.

3.5 Evaluation

I 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?
- 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 §3.5.2 and §3.5.3, I 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. I run the client on server-class machines within the same rack as my 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 §3.5.4 and running my ported apps in §3.5.5, I interchangeably use Samsung Galaxy Nexus phones and an Asus Nexus 7 tablet, all running Android 4.2 and my sClient. In §3.5.5, I discuss how I use Simba to enable multi-consistency and to fix inconsistency in apps. I do not verify correctness under failures, as this is handled in [37]. sCloud consists of around 12K lines of Java code counted using CLOC [5] as shown in Table 3.6.

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

Table 3.6: Lines of code for sCloud. Counted using CLOC.

3.5.1 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, I want to ensure that Simba does not add significant overhead to the sync process. Thus, I first demonstrate that Simba's sync protocol is lightweight. To do so, I measure sync overhead of rows with 1 byte of tabular data and (1) no, (2) a 1 byte, or (3) a 64 KiB object. I generate random bytes for the payload in an effort to reduce compressibility.

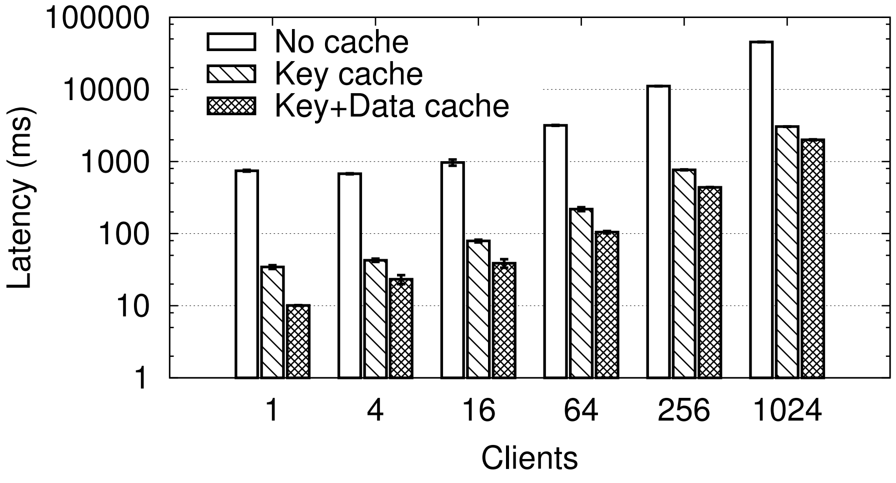
# of Rows	Object Size	Payload Size	Message Size (% Overhead)	Network Transfer Size (% Overhead)
1	None	1 B	101 B (99%)	133 B (99.2%)
	1 B	2 B	178 B (98.9%)	236 B (99.2%)
	64 KiB	64 KiB	64.17 KiB (0.3%)	64.34 KiB (0.5%)
100	None	100 B	2.41 KiB (96%)	694 B (85.6%)
	1 B	200 B	9.93 KiB (98%)	9.77 KiB (98%)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

Table 3.7: 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.

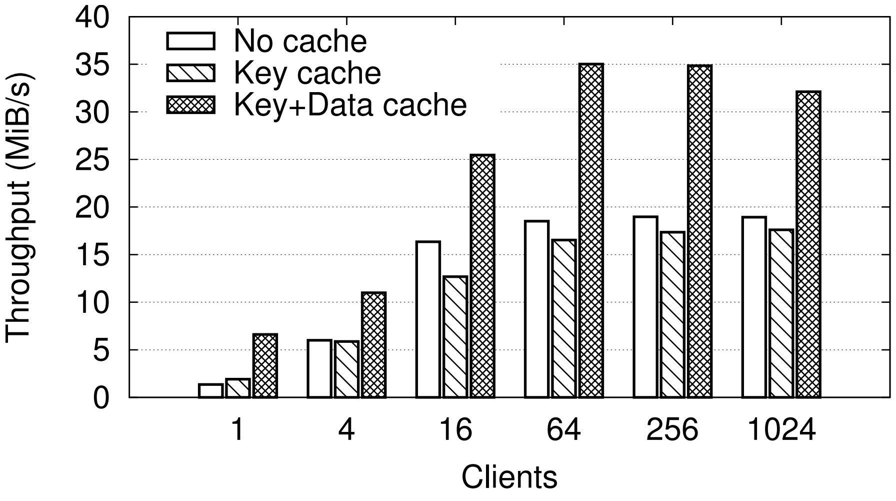
Table 3.7 quantifies the sync protocol overhead for a single message containing 1 row (*i.e.*, worst-case scenario), and a batch of 100 rows. For each scenario, I show the total size of the payload, message, and network transfer. I find that the baseline (no object data) message overhead for a single 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, my results show that Simba Sync Protocol is indeed lightweight.

3.5.2 Performance of sCloud

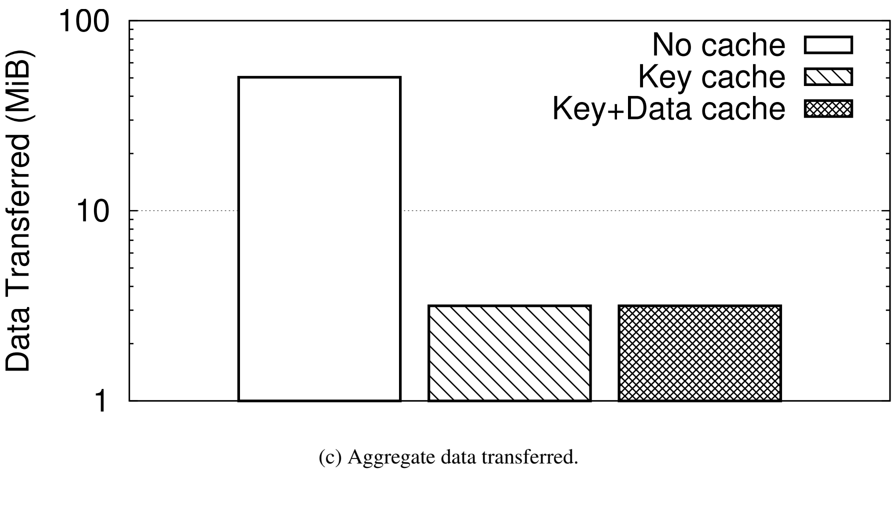
In this section, I evaluate the upstream and downstream sync performance through a series of microbenchmarks. The intent of these benchmarks is simply to measure raw performance of the system under a variety of settings. I provision 48 machines from PRObE's Kodiak testbed [36]. Each node is equipped 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. I deploy an sCloud configuration with a single Store node and a single gateway. I configure Cassandra and Swift on disjoint 16-node clusters for backend storage; the remaining 16-nodes are used as client hosts. I use my 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, I set the compressibility of object data to be 50% [44].



(a) Per-operation latency.

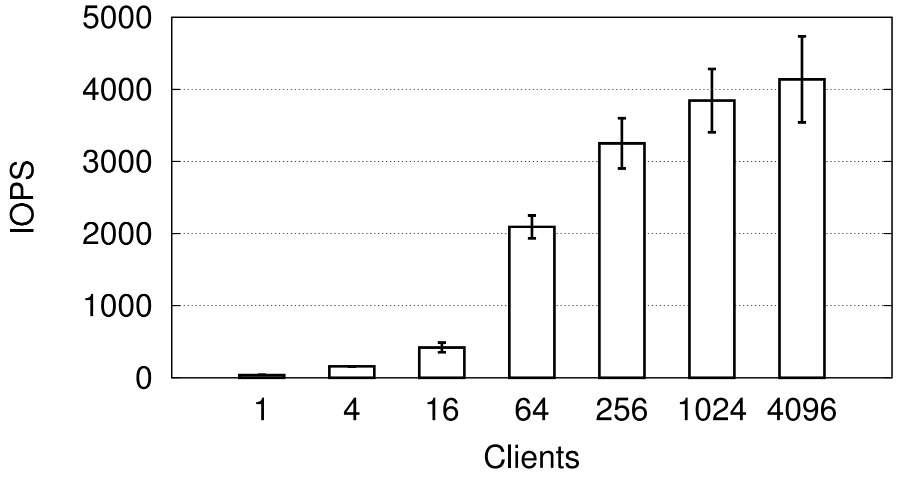


(b) Aggregate node throughput.

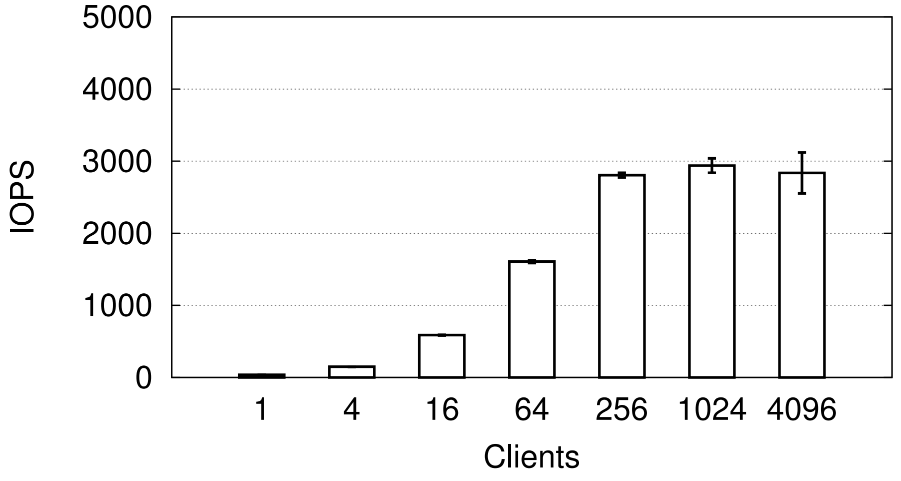


(c) Aggregate data transferred.

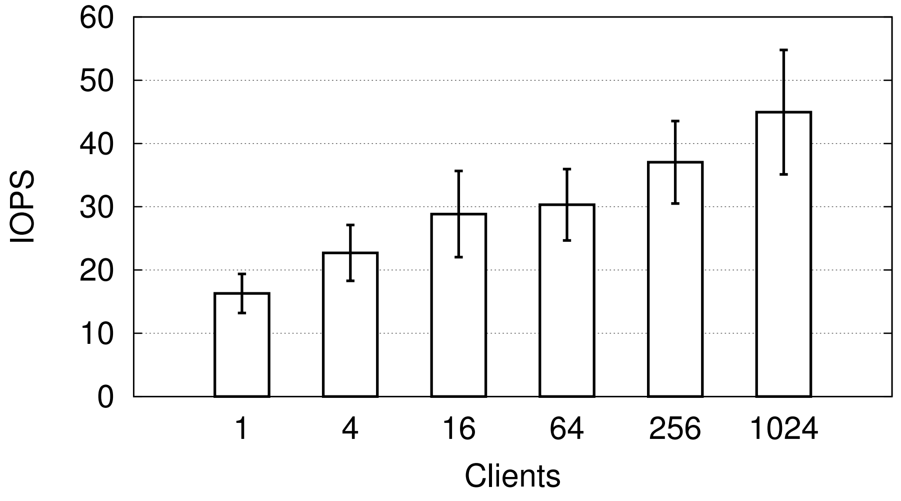
Figure 3.4: Downstream sync performance for one Gateway and Store. Note logscale on y-axis in (a) and (c).



(a) Gateway only.



(b) Store with table-only.



(c) Store with table+object.

Figure 3.5: Upstream sync performance for one Gateway and Store.

3.5.2.1 Downstream Sync

My 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 my change cache (described in §3.4).

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 3.8: Server processing latency. Median processing time in milliseconds; minimal load.

I 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 3.8. In order to populate the Store with data, a writer 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. I 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 3.4(a) shows the client-perceived latency as I 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 3.4(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, I 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 3.4(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.

3.5.2.2 Upstream Sync

Next, I evaluate the performance of upstream sync. I 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. I 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.

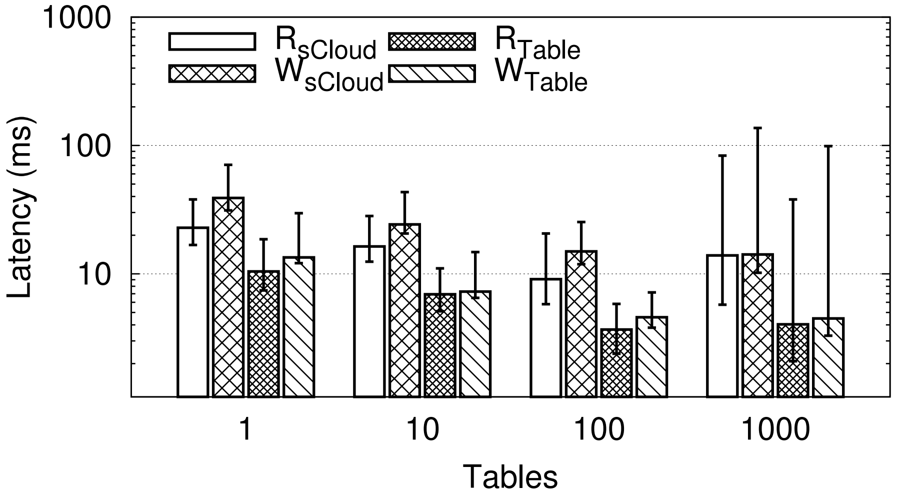
Figure 3.5 presents the client-perceived upstream performance. The first test stresses the Gateway alone by sending small control messages which the Gateway directly replies so that

Store is not the bottleneck (Figure 3.5(a)). The figure shows that Gateway scales well up to 4096 clients. In the second test, I wrote rows with 1 KiB tabular data and no objects. This exercises Store with Cassandra but without Swift. As shown in Figure 3.5(b), Store performance peaks at 1024 clients, after which Cassandra latency starts becoming the bottleneck. For the third test, I wrote rows with 1 KiB of tabular data and one 64 KiB object. This exercises Store with both Cassandra and Swift. In Figure 3.5(c), I 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.

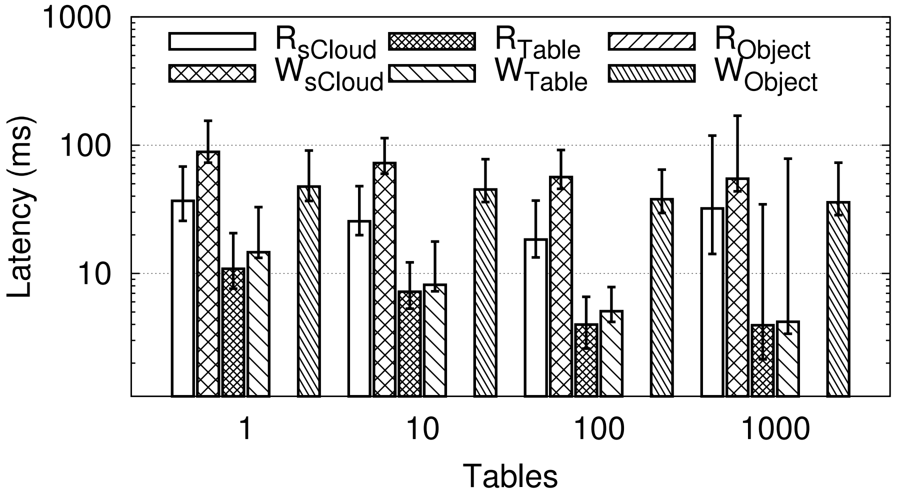
3.5.3 Scalability of sCloud

I now demonstrate the ability of sCloud to be performant at scale when servicing a large number of clients and tables. I provision 32 nodes from the PRObE Susitna cluster [36]. Each node is equipped with four 16-core AMD Opteron 2.1GHz CPUs, 128GB DRAM, two 3TB 7200RPM data disks, and an InfiniBand high-speed interface. I 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.

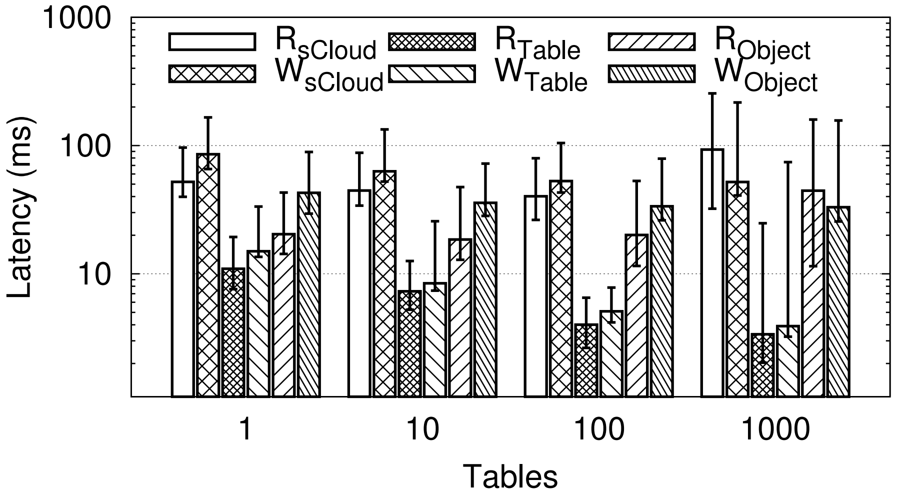
I scale the number of tables and clients, and use the Linux client to instantiate multiple clients partitioned evenly across up to 16 nodes. Each client has either a read or a write subscription to a particular table. I set the ratio of read-to-write subscriptions as 9:1 and partition them evenly across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



(a) Table only.



(b) Table+object, chunk cache enabled.



(c) Table+object, chunk cache disabled.

Figure 3.6: sCloud performance when scaling tables.

3.5.3.1 Table Scalability

I 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. I set the number of clients as 10x the number of tables.

Figure 3.6 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 3.6(a)), I 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 3.6(b)) and without (Figure 3.6(c)) the chunk cache, I observe a similar decreasing trend in the Cassandra latency, and to a lesser extent, in the Swift write latency, as I 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. I 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, I observe correlated latency spikes for Cassandra and Swift, suggesting the backend storage to be the culprit.

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 3.9: sCloud throughput at scale.

Table 3.9 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.

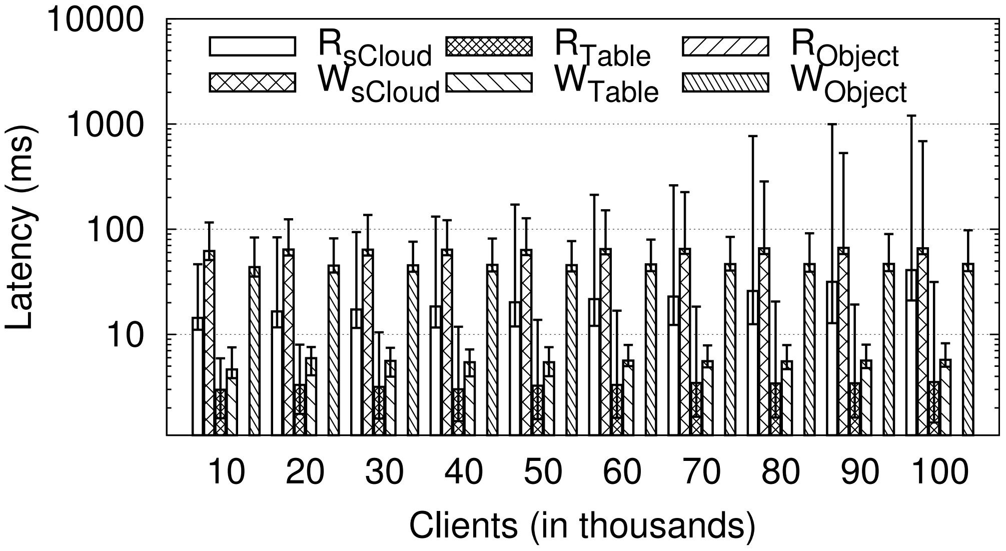


Figure 3.7: sCloud performance when scaling clients.

3.5.3.2 Client Scalability

In the previous case, I found that scaling the number of tables in Cassandra leads to significant performance overhead. Thus, I also investigate sCloud's ability to scale clients with fewer tables. Figure 3.7 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 than 100 ms. Tail latency increases as I scale which I attribute to increased CPU load. Overall, I find that sCloud scales efficiently under high client load.

3.5.4 Trade-off: Consistency vs. Performance

On two Samsung Galaxy S3 phones running Android 4.2, I 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 [85] using dummynet [32] to sCloud.

I 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

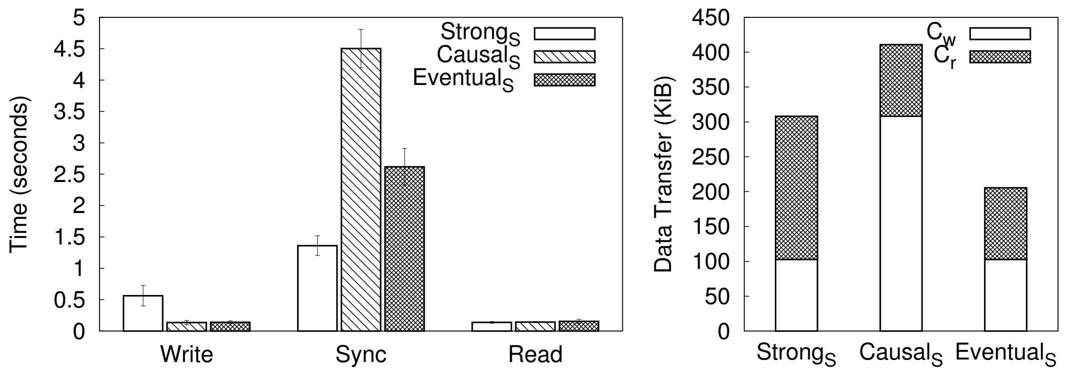


Figure 3.8: Consistency comparison. End-to-end latency and data transfer for each consistency scheme.

the sCloud, under each of my 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, I 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. I 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 3.8 shows the WiFi latency and associated data transfer (3G results similar; not shown). I 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". I 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.

3.5.5 Writing Simba Apps

Writing a multi-consistent app: I used an existing app, Todo.txt [9], 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. I 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 my 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, I ported UPM to use Simba. I tried two approaches:

- Store the entire account database as an object in a sTable. This required fewer modifications; I 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 work a total of a few hours (< 5) to port UPM to Simba through both the above approaches.

3.6 Summary

Quoting Butler Lampson: *“the purpose of abstractions is to conceal undesirable properties; desirable ones should not be hidden"* [52]; 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, I 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, I 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. I have built Simba, a data-sync service that provides app developers with an easy-to-use yet powerful sTable-based API for mobile apps; I 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`.

Chapter 4

Lego: A Modular Emulator of Distributed Object Stores

Over the last decade, a large number of distributed object stores have been developed, each of which export a unified GET/PUT interface to data spread across the servers in a data center. Notable examples include GFS [35] to store web crawls at Google, Dynamo [27] to store shopping carts at Amazon, and Haystack [18] to store photos at Facebook.

This preponderance of distributed object stores partly stems from the need to tailor an object store's design to the workload characteristics and hardware configurations in the environment where the system is to be deployed. For example, GFS [35] chooses to split objects into large 64 MB chunks and uses chain replication due to the need to support reads and writes of gigabyte-sized files at high throughput. Similarly, FAWN [16] stores only a fragment of every key in in-memory metadata due to the limited memory capacity on embedded devices. An inappropriate design can either make it impossible to satisfy service-level objectives (SLOs) for throughput and latency or significantly increase the number of storage servers necessary to do so.

Due to this dependence of object store design on workload and hardware characteristics, as new workloads emerge and hardware configurations evolve, the need for new object stores is likely to continue. Developing a production-quality object store is however an arduous task that can take several man-years of work. Therefore, it is critical that developers make a carefully informed choice for the design of the new system before beginning development.

To aid developers in choosing the system design that is best suited to a new deployment setting, I design and implement Lego, a distributed object store emulator. Though not equivalent to a production-quality system (e.g., it does not handle failures), the Lego emulator enables developers to measure and compare the performance offered by various alternatives for data and metadata management. Since Lego enables deployment and measurement on a real cluster, unlike simulations, it captures the impact on performance due to hardware and workload peculiarities (e.g., latency effects caused by variations across disks [50] or TCP incast [74] caused by limited buffers in switches).

To enable rapid prototyping of storage system designs, Lego uses a modular architecture, which minimizes developer overhead by enabling *reuse of modules* across storage systems. Such reuse is possible due to the significant similarities that often exist across different designs, e.g., two designs may store data in an identical manner on disk but may differ in their metadata management, or may use the same replication strategy but use a different policy for selecting replicas. However, designing a modular framework for distributed object stores that enables module reuse is associated with several fundamental challenges.

- **Generality.** For modules developed as part of any object store to be reusable across systems, Lego's architecture must enable modules to be combined in such a manner that, though the combination successfully facilitates the execution of PUTs and GETs, every module is oblivious to the implementation details of other modules. This poses a trade-off between generality and reuse; more fine-grained modules will enable better reuse, but may not be inter-operable with all implementations of other modules.

- **Partial reuse.** When coarse-grained modules are used to enable broader reuse, it will often be necessary for different storage systems to reuse *parts* of module implementations, rather than reusing modules in their entirety.
- **Locality.** Lastly, module implementations often make implicit assumptions of locality, and these assumptions can be violated when a particular implementation of a module is coupled with certain implementations of other modules. For example, though one component may be writing data sequentially to disk, these writes may translate to random I/O when another component reads and writes metadata from random locations on the same disk.

In this chapter, I describe my design of Lego and present results from my experience of using it to instantiate 10 popular distributed object stores. My design requires any object store to be developed as a set of event handlers, where the execution of any handler is triggered at any node in the cluster upon the receipt of a specific type of message at that node, and at the end of its execution, each handler outputs a message that leads to the execution of other event handlers. Every event handler itself is composed of a DAG of stateful functions, each of which relies only on the interfaces exported by its neighbors in the DAG but not on how they are implemented. Thus, code written for any distributed object store can be reused either at coarse event handler granularity or at the finer granularity of stateful functions.

I evaluate Lego's benefits compared to a scenario wherein every object store is implemented independently. By imposing on developers a way of implementing distributed object stores that enables code reuse, I find that Lego reduces the total number of lines of code required to implement 10 popular distributed object stores by over 80% (~17K lines of code). Despite such reuse, Lego preserves the rank ordering of object store designs in terms of their ability to satisfy performance goals cost-effectively by imposing low performance overhead; on average across the 10 systems, it increases 99^{th} percentile by 12%.

While I focus on my modular object store emulator in this chapter, my approach for developing Lego can have broader implications. The current undesirable scenario of having to implement a new storage system from scratch for every new deployment setting stems from the fact that every system has assumptions about its target setting baked into its implementation. As a result, it is often non-trivial to modify any existing system in order to support new workload requirements or hardware characteristics. Thus, beyond simplifying the task of choosing the right design of a distributed object store, I envision that the principles underlying Lego's modular approach can also be applied to enable the development of evolvable system implementations.

4.1 Motivation

My motivations for Lego are two-fold: the inadequacy of any single object store for all settings, and the inflexibility of the implementations of production-quality storage systems.

4.1.1 Importance of tailoring object store design

First, I demonstrate why it is the case that a large number of distributed object stores have been developed, and why more storage systems will be necessary in the future. I use three object stores—HDFS [45], Voldemort [99], and Haystack [18]—that have been designed for disparate workloads. HDFS, which is based on GFS's [35] design, is well-suited for storing and reading large objects at high throughput. Voldemort mimics the design of Dynamo [27], which was designed to serve GETs/PUTs on small objects with low latencies. Lastly, Haystack [18] is optimized to enable low-latency reads on small write-once read-many objects. Since Haystack's code is not public, I use Bitcask [21] to build Haystack*, an imitation of Haystack's design.

Workload	**Object Size**	**Read/Write Characteristics**	**Performance SLO**
Workload1	640 MB	Read-many write-once	0.1 PUTs/s, 1 GET/s, Median GET latency < 1s
Workload2	4 KB	Read-many write-many	20 PUTs/s, 20 GETs/s, 99^{th} %ile GET latency < 100ms
Workload3	64 KB	Read-many write-once	10 PUTs/s, 30 GETs/s, 99^{th} %ile GET latency < 100ms

Table 4.1: Example workloads.

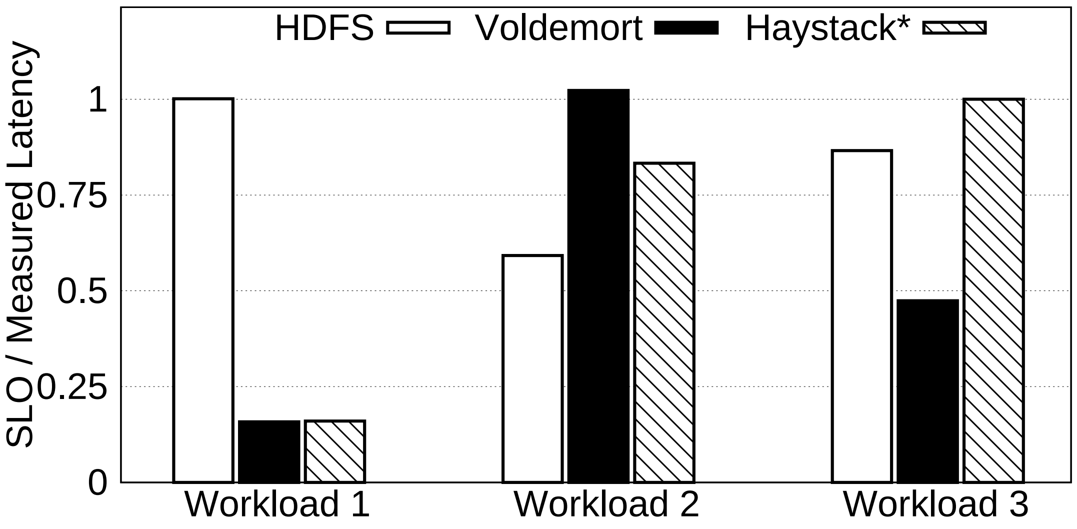


Figure 4.1: Comparison of three object stores with respect to their ability to meet the SLOs in three different workloads.

I test the performance offered by these systems on the workloads for which they were designed (described in Table 4.1). Though my workloads are relatively simplistic (e.g., no skew across objects either in terms of size or read/write rates), these suffice for my purpose of demonstrating the variance in a particular system's performance across workloads. For every (object store, workload) pair, I deployed the object store on a cluster of servers (each with eight 7.2K-RPM disks and 24 GB RAM) on a shared 10 GbE network, and measured the GET latencies when subjected to the throughput specified in the corresponding workload. I keep hardware costs comparable across deployments of the three systems.

Figure 4.1 compares the ability of each object store to meet the specified SLOs. For each (object store, workload) pair, I plot the ratio of the GET latency SLO for that workload to the corresponding percentile of GET latencies measured when using that object store to serve the workload. For example, the bar for Voldemort on *Workload2* is the ratio of 100 ms (SLO in *Workload2*) to the 99^{th} percentile GET latency when using Voldemort for *Workload2*. When this ratio is $>= 1$ (< 1), it indicates that the GET latency SLO for the corresponding workload was met (not met).

Though the cost of hardware was the same in our deployments of all three object stores, I see that the ability to meet the SLOs associated with a particular workload varies significantly across object stores. For example, in *Workload1*, HDFS's design of splitting objects into large chunks makes it a good match for reading large objects; in contrast, since Voldemort and Haystack* are designed to store small objects, they do not split objects into chunks, thus causing the read of a large object to be limited by the bandwidth of any single disk. Similarly, storing on-disk locations of all objects in memory enables Haystack* to deliver low latencies in *Workload3*, whereas HDFS does not cache metadata since it has been designed to optimize throughout, not latency. In all cases where the SLOs were not met, either the system's design makes it fundamentally impossible to satisfy the SLO or a larger storage cluster is necessary.

These results illustrate that the design of a distributed object store that can most cost-effectively satisfy performance SLOs is strongly dependent on the workload and its requirements. Though not shown here, the cost-optimal design of an object store is also similarly dependent on the hardware configuration of the cluster on which it is deployed [16, 59].

4.1.2 Challenges in evaluating object store designs

While it is important to tailor the design of an object store to the characteristics of the environment in which it is to be deployed, evaluating the performance that any particular design will offer is not straightforward.

On the one hand, one cannot estimate the performance that a particular design will offer simply via back-of-the-envelope calculations. For example, based on the manner in which a design stores and retrieves data and metadata on hard disks, SSDs, and RAM, I can attempt to estimate the performance offered by any particular scale of deployment by accounting for workload characteristics (e.g., object size distribution and GET:PUT ratio) and micro-benchmarks of hardware performance (e.g., disk seek times and network bandwidth). However, such an estimate will ignore delays introduced due to queueing both in the network and at storage servers. Moreover, it is hard to capture all the intricacies of storage hardware using simple micro-benchmarks, e.g., there are significant variances across different units of the same type of hard disk drive [50], and SSD performance depends on its firmware, whose logic can be hard to model. Therefore, it is important to capture the *dynamic interactions* between the input workload and the target hardware, in order to accurately estimate performance.

On the other hand, it is hard to evaluate a new object store design simply by modifying the implementation of an existing system. This is because the implementations of most object stores are not amenable to changes to their design. Storage system developers typically focus their implementation efforts on ensuring the best performance, reliability, and consistency in operational deployments; their focus is not as much on ensuring extensibility. Thus, every system implements each of its components in one particular manner—that which is best suited for the system's target setting—and though these systems expose several configuration parameters and tweaking these parameters can significantly vary performance [46], the range of parameters

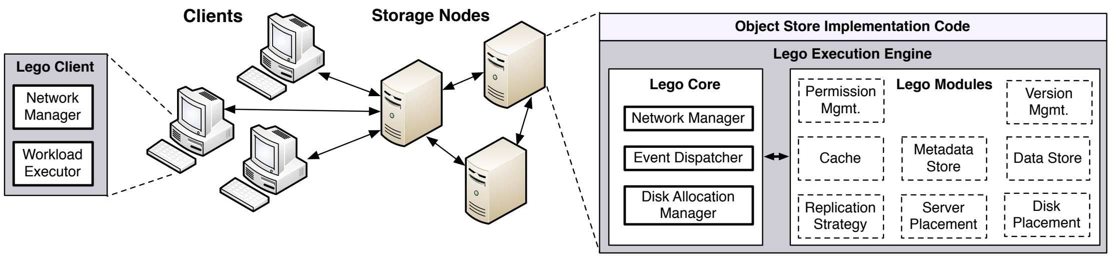


Figure 4.2: Logical system architecture of Lego. In the deployment of any object store, every machine either serves as a client or server or both. On every server-side machine, the Lego execution engine invokes modules that collectively comprise the object store's implementation.

is limited. For example, to minimize latencies, neither does HDFS permit explicit caching in memory of on-disk locations of chunks, nor does Voldemort allow metadata to be centralized.

4.2 Lego Overview

To ease the evaluation of the cost vs. performance trade-off offered by any particular design of a distributed object store, I have designed and implemented the Lego emulator. Lego enables a developer to instantiate different designs, deploy them on a cluster that is representative of the hardware configuration in the target setting, and measure the performance obtained when the system is subjected to an appropriately scaled down version of the target workload, all at a significantly reduced effort compared to the current status quo. Though the measured performance may not accurately reflect a production-quality implementation at scale, I posit that emulation with Lego offers sufficient fidelity to preserve the rank ordering of system designs in terms of their ability to satisfy performance goals cost-effectively. For this, I design Lego to enable the faithful reproduction of the manner in which any particular design stores data and metadata, and executes reads and writes.

In this section, I describe my high-level approach in designing Lego and the associated challenges.

4.2.1 Approach

My design of Lego is guided by my goal to reduce developer effort when instantiating a prototype of any distributed object store. To meet this goal, my key insight is to leverage the commonalities that often exist across the designs of different object stores. Here are a couple of examples of such similarities between systems.

Consider GFS [35] and Haystack [18], two existing object stores which target workloads with very different performance goals. As I observed previously, these systems significantly differ in the manner in which they store data on disk due to the differences in their target workloads and SLO requirements. However, both systems use a central directory which stores a mapping for each object to the servers at which it is stored. Therefore, to simplify the instantiation of Haystack's design, it would be desirable to reuse the central directory implementation from GFS's instantiation.

Similarly, I observe the potential for reuse when comparing chain replication [98] and CRAQ [88]. Both systems offer strong consistency, albeit in different workload settings: chain replication focuses on offering high write throughput, whereas CRAQ seeks to minimize read latencies. However, in either system, a PUT operation is relayed along a sequence of nodes, with each node passing on the update to its successor once it has committed the update locally. Thus, despite other differences between the two designs (e.g., unlike chain replication, once a PUT is committed at the tail, CRAQ propagates an acknowledgement back along the sequence of nodes), there exists significant similarity between the two that offers the potential for code reuse.

To exploit such similarities between storage systems, I design Lego such that it enables any object store to be implemented as a composition of modules. Every module either enforces policy or stores state, in combination facilitating the execution of reads and writes. Lego's modular architecture enables rapid prototyping because any new system can be instantiated by reusing modules written as part of systems previously instantiated in the Lego framework; indeed, as I show later in Section 4.5, I find that some systems can be constructed entirely by reusing code from previously written systems. While the idea of a modular architecture is not a new one—there are several well-known examples in previous work [62, 33] which show the benefits of such an approach—I apply modularity to ease the emulation of distributed object stores.

4.2.2 Challenges

In order for modules to be reusable across object stores, every module should be oblivious to the implementation of other modules. This requirement poses two significant challenges, which I discuss here.

Granularity of code reuse. In many cases, two object stores may require *similar* but *not identical* implementations of a particular component. As an example, consider the similarity between chain replication [98] and CRAQ [88], as described above. In both, updates are relayed from the head to the tail in the replica chain. However, the two designs differ in how the tail node in a chain operates; in CRAQ, the tail node sends an acknowledgment back along the chain, whereas in chain replication, the tail node directly acknowledges the write to the client. In such cases, the granularity at which modules are implemented critically determines the amount of reuse possible. For example, if the code that runs at every storage server is a module, then the code for CRAQ will not be reusable in chain replication, and vice-versa. Whereas, if modules are too fine-grained, the line between original development and code reuse blurs, and the developer does not save much implementation effort.

Locality. When multiple modules perform I/O to the same disk or SSD, locality assumptions can be violated. For example, the manner in which data stored by a data store is spread across a disk can vary depending on how the co-located metadata store instances store data on the same disk. Without proper care, competing modules may turn what is intended to be sequential I/O into random I/O. Also, to ensure the modularity of Lego, it is key that the data store for every disk need no knowledge about how co-located metadata store modules are implemented or deployed.

4.3 Lego Design

A developer seeking to use Lego to emulate the operation of any particular design for a distributed object store must take two steps. First, she must implement the object store in the manner specified by Lego, potentially reusing code from object stores previously implemented using Lego's framework. Second, she uses Lego to deploy, execute, and evaluate the performance of the object store on a cluster of machines. In this section, I describe the development framework prescribed by Lego and the execution engine that it uses to support the operation of any system.

Recall that my goal is not to mimic the performance at scale or fault tolerance offered by a production-quality object store implementation. Instead, I seek to ease the comparison of cost vs. performance trade-offs across different system designs via rapid prototyping.

4.3.1 Development framework

My main goal in designing the manner in which object stores must be implemented is to enable code reuse when there exist similarities between systems, like the examples described earlier in Section 4.2.1. To meet this goal, I observe that the set of modules that are combined to construct any object store must together answer the fundamental question which any storage system design must: *"how does one perform reads and writes?"* For this, in each system, some of the modules must encode *policies* of how read and write operations are executed, and other modules must maintain *state* in the form of metadata and data. The modules running on different machines in the cluster then coordinate with each other to facilitate end-to-end execution of reads and writes.

4.3.1.1 Coarse-grained reuse

Event handlers. Given my abstract model of any distributed object store as described above, I represent every object store implemented in Lego as a set of event handlers; when Lego is used to emulate the operation of any object store, every machine in the cluster is continually executing event handlers in response to corresponding events. I make event handlers as the primary building block in Lego because we observe that every machine in an object store's deployment has work to do only in one of three scenarios: 1) when asked to do so by a client (e.g., looking up the set of replicas for an object), 2) when work is delegated to it by another server (e.g., updates to replicas being propagated along a chain), or 3) when certain conditions about the system's state are satisfied (e.g., run garbage collection when disk is full or periodically check if other machines are up). The machines in the cluster can collectively serve reads and writes because the execution of an event handler on any one machine can lead to the invocation of other event handlers either locally or on other remote machines.

Events in Lego that can lead to the invocation of event handlers can be of two types: receipt of messages and invocation of triggers.

Messages. Every message specifies a type and a destination machine (on which event handler to process this message is invoked), and includes a set of parameters. A separate handler is associated with every message type. When any machine receives a message and the event handler associated with that message type is invoked, the handler can make use of the parame-

Lookup(key)

LookupResponse(key, replicas[])

Put(key, data, version)

PutResponse(key, version, msg, success?)

Get(key, version)

GetResponse(key, data, msg, success?, version)

PutMetadata(key, data, version)

PutMetadataResponse(key, msg, success?)

GetMetadata(key, version, versionOnly?)

GetMetadataResponse(key, data, version, msg, success?)

Table 4.2: Example set of message types currently defined and available in Lego from current system implementations.

ters included in the message. At the end of its execution, an event handler may emit messages directed either at the local or at remote machines, causing other event handlers associated with those message types to be invoked at the corresponding machines.

While every system implementation is free to define its own message types and formats, Table 4.2 shows some of the message types that are common across the 10 distributed object stores that I have implemented thus far with Lego.

Triggers. In addition to invoking event handlers based on the receipt of messages, Lego also supports event handler invocations based on asynchronous events, which I call triggers, outside of the system serving reads and writes. I currently support two types of triggers. On the one hand, an object store can register for time-based triggers, e.g., synchronize with a backup every 30 seconds or run an event handler 3 seconds from now to account for timeouts. On the other hand, Lego supports system-state based triggers (e.g., run garbage collection when disk space utilization goes above 80% or flush a table in memory to disk when memory utilization exceeds

Code 1 Event handlers for handling PUTs in CRAQ

```
procedure HandlePutRequest(request, chain, reverse_chain)
    writeToDisk(request.key, request.data)
    if thisNode == chain.tail then
        response = generatePutResponse(request)
        sendPutResponse(response, reverse_chain.next)
    else
        forwardPut(request, chain.next, reverse_chain)
procedure HandlePutResponse(response, chain)
    updateVersion(response.key, response.version)
    forwardPutResponse(response, chain.next)
```

a threshold) that an object store can register for when initialized.

Note that, in any cluster deployment of an object store, different machines serve different roles, e.g., metadata directory vs. storage server. Therefore, the set of event handlers can differ across machines. For example, upon receiving a GET message that specifies an object's name, a node that serves as the metadata directory may run an event handler which looks up the replicas storing the object, whereas a storage server that receives the same message will attempt to read the object from local disk. The type and contents of the output message will also differ in these cases.

Reusing event handlers. Representing every object store as a set of event handlers naturally enables code reuse; an implementation of any object store design can reuse event handlers written by previously implemented systems.

For example, consider the pseudocode shown in Code 1 for the two event handlers that CRAQ must implement at the storage servers hosting replicas of data. Upon receiving a PUT message, a storage server executes *HandlePutRequest*, which forwards the PUT to the next node along the chain unless this node is the last node on the chain; if this node is the tail, it sends a PutResponse message to the previous node on the chain. When any node receives a PutResponse message, *HandlePutResponse* further propagates this message back up the chain. If a client C writing to an object stored on replicas R_1, R_2, and R_3 initializes the *chain* in its initial PUT message as (R_1, R_2, R_3) and *reverse_chain* as (R_3, R_2, R_1, C), then in combination, *HandlePutRequest* and *HandlePutResponse* ensure that the data being written is propagated along the chain and the response is propagated back to the client.

Given this code for an implementation of CRAQ, implementing chain replication becomes straightforward. One can simply reuse the *HandlePutRequest* event handler, without the need for using *HandlePutResponse*. Only a change to the client is necessary; when a client initiates a PUT in this case, it must initialize *chain* parameter as before but set *reverse_chain* to (R_3, C) so that the tail of the chain directly acknowledges the PUT to the client.

Note that, for simplicity, the pseudocode in Code 1 does not depict error handling. In my implementation of CRAQ and chain replication, I do account for the need for intermediate nodes along a chain to send responses when operations fail.

Client. In the Lego development framework, clients are not based on event handlers; instead, every client runs a set of threads (the number depends on the desired throughput), where each thread issues reads and writes based on either an input workload trace or a workload characterization. For every read/write, the client correspondingly emits GET and PUT messages. Depending on an object store's design, the client may have to interact with a single front-end storage server or may have to distribute its requests across a set of nodes (e.g., using consistent hashing). The client can also store and update object version metadata for implementations that choose to do so.

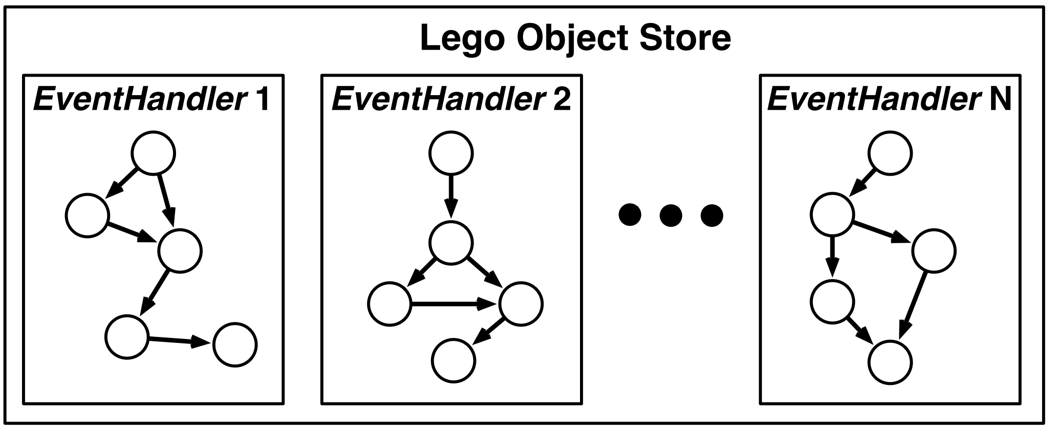
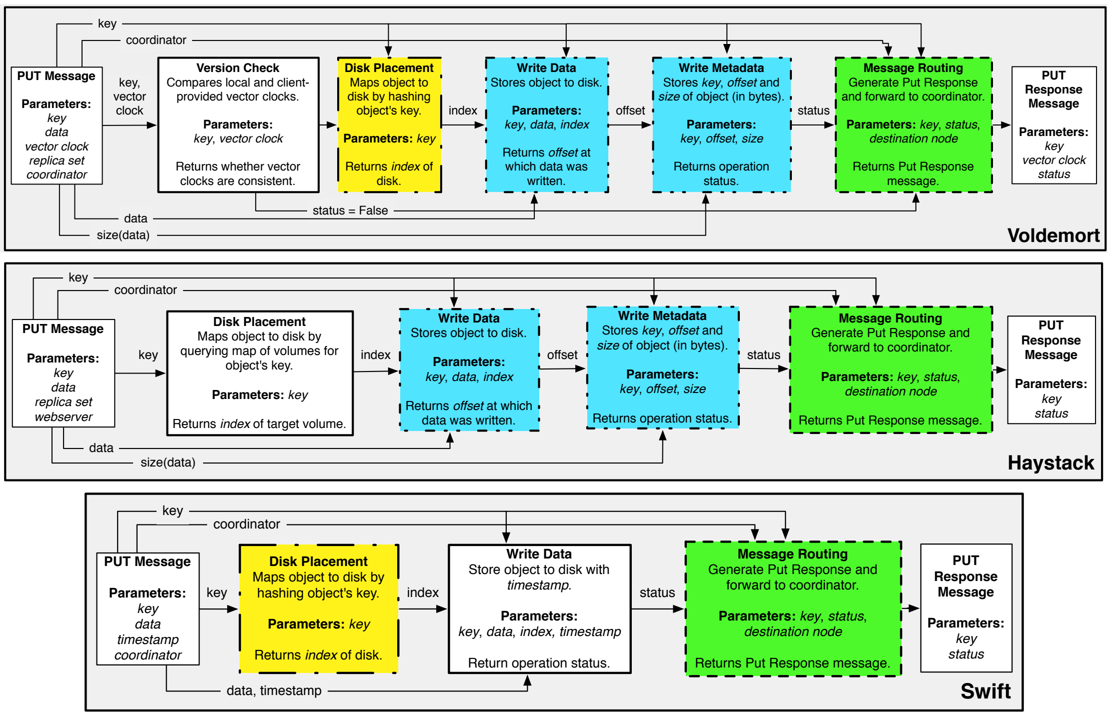


Figure 4.3: An object store implementation in Lego comprises multiple event handlers, each of which is implemented as a DAG of stateful functions.

4.3.1.2 Fine-grained reuse

Reusing code from a previously implemented system by simply reusing event handlers from that system is not always possible. Since the code for an event handler itself can be complex, reusing code at the granularity of event handlers limits the potential for reuse. For example, when handling PUTs, storage servers in both Voldemort and Swift are identical in terms of mapping an object to one of the local disks, but they differ significantly in terms of how they store data and metadata on disk.

Stateful functions. To enable reuse in such cases, I observe that any event handler can itself be broken down into smaller modules, where each module is responsible either for enforcing policy or storing state in a specific manner. Thus, as shown in Figure 4.3, every event handler in Lego can be represented as a set of inter-connected modules, where each module receives input from some modules and provides input to other modules. Broadly, the modules within any event handler form a directed acyclic graph (DAG), where an initial module is executed when the handler is invoked, which in turn leads to the executions of other modules, finally resulting



key
coordinator
PUT Message
Parameters: key data vector clock replica set coordinator
key, vector clock
Version Check
Compares local and client-provided vector clocks.
Parameters: key, vector clock
Returns whether vector clocks are consistent.
Disk Placement
Maps object to disk by hashing object's key.
Parameters: key
Returns index of disk.
index
Write Data
Stores object to disk.
Parameters: key, data, index
Returns offset at which data was written.
offset
Write Metadata
Stores key, offset and size of object (in bytes).
Parameters: key, offset, size
Returns operation status.
status
Message Routing
Generate Put Response and forward to coordinator.
Parameters: key, status, destination node
Returns Put Response message.
PUT Response Message
Parameters: key vector clock status
data
size(data)
status = False
Voldemort
PUT Message
Parameters: key data replica set webserver
Disk Placement
Maps object to disk by querying map of volumes for object's key.
Parameters: key
Returns index of target volume.
Returns Put Response message.
PUT Response Message
Parameters: key status
Haystack
PUT Message
Parameters: key data timestamp coordinator
Write Data
Store object to disk with timestamp.
Parameters: key, data, index, timestamp
Return operation status.
data, timestamp
Swift

Figure 4.4: Reused modules in the event handlers for PUTs across three object store implementations in Lego. Identical modules across systems have the same color and border.

in the output of messages and/or the registration of triggers.

Since every module is akin to a function, albeit that it stores state across invocations of the event handler that uses this module, I refer to these modules as *stateful functions*. The key property that Lego imposes on these modules is that each module only be dependent on the interfaces exported by other modules but be agnostic to their implementation. This property ensures that, when new workloads or hardware configurations emerge, new system designs that are necessary to satisfy the requirements of new settings can potentially reuse modules developed previously. Over time, as more systems are implemented using Lego, the number of available module implementations that a developer can draw upon will grow, thus simplifying the task of constructing a system with a new design.

Examples of reuse. Figure 4.4 shows how three different object stores—Voldemort, Haystack, and Swift—can reuse each others' modules in their event handlers for the PUT message. For each system, I denote the modules which are fully reused across these implementations by using a common color and border. Voldemort and Swift share the same module for *Disk Placement* (shown in yellow with a dash-dot pattern border), whereas Voldemort and Haystack share the same modules for *Data Store* and *Metadata Store* (shown in blue with a dash-dot-dot pattern border). All three systems share the same *Replication Strategy* (shown in green with a dash pattern border). This example shows that, across different systems, the implementation of the event handler for the same event can share some modules but not all.

Enabling finer-grained code reuse is also useful when the code for an existing implementation of a module can be reused as is, but executed in a different manner. For example, though both Cassandra and Haystack maintain metadata as an in-memory map, the former commits the metadata to disk when memory usage exceeds a threshold whereas the latter does the same periodically. In this case, the *Metadata Store* module in Cassandra can be reused to realize Haystack's design, albeit with the event handler that commits metadata to disk associated with a different trigger.

4.3.2 Execution engine

Once a developer has implemented an object store as described above—a set of event handlers, each of which is a DAG of stateful functions—the Lego execution engine enables the operation of the object store on a cluster of machines. On every machine in the cluster, the Lego execution engine receives messages and invokes triggers, executes event handlers in response to these, provides the interface that modules can use to interact with hardware, etc. It does so via three main components; I describe each of them next.

The **Network Manager** handles all network communication between Lego modules and facilitates the transport of messages between clients and servers, or amongst servers. It also handles the marshalling and unmarshalling of messages, respectively, before they are sent out

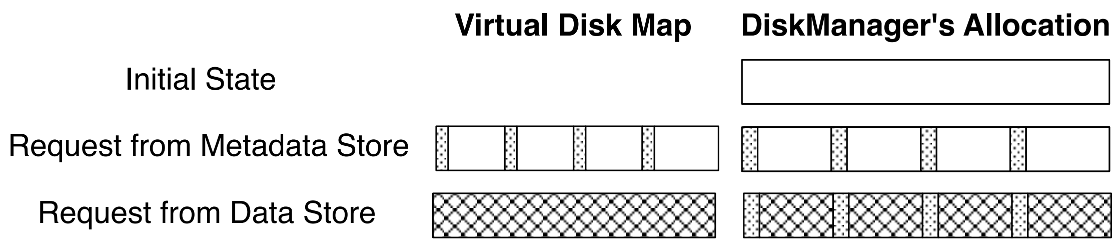


Figure 4.5: Illustration of the manner in which the Disk Manager can manage disk space allocation requests.

over the network and before they are handed as input to an event handler. Note that a message emitted by an event handler executed on one machine could potentially be destined for the same machine, e.g., when a server that coordinates the execution of a read issues GETs to all replicas, one of the replicas of the object could be stored on the local node.

The **Event Dispatcher** keeps track of when events occur and invokes the associated event handlers. To do so, on startup, the Event Dispatcher on any machine receives the set of time-based and state-based triggers that it must track and the set of event handlers that it must use. Thereafter, during execution, the Event Dispatcher on every machine receives messages from the local Network Manager.

Lastly, the **Disk Manager** interfaces between every module's interactions with storage, with one instance per disk. To enable sharing of disk space and I/O among modules, the Disk Manager presents an abstract virtual interface to disk space and translates operations on the virtual disk to corresponding physical disk.

On startup, every module intending to allocate disk space must issue a *DISK_ALLOC* request to the Disk Manager of every disk that it seeks to use. *DISK_ALLOC* requests specify a bit-vector (with one bit per disk sector) which represents the configuration in which the requester wishes to reserve space on disk. Every Disk Manager then reconciles all the *DISK_ALLOC* requests that it receives; Figure 4.5 presents an illustrative example. Thereafter, when any module needs to read from or write to a disk, it issues this operation to the Disk Manager for that disk,

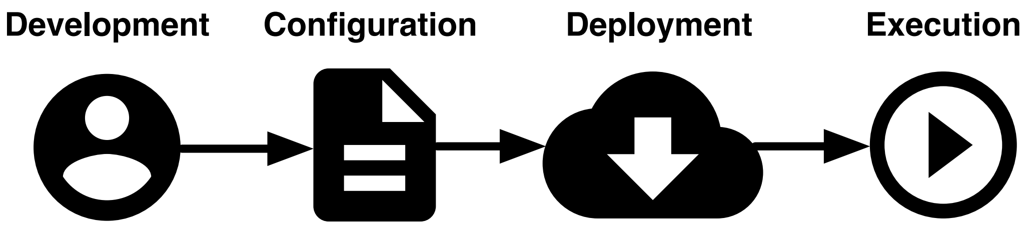


Figure 4.6: Stages of the Lego life-cycle.

which in turn relays the I/O after transforming the operation's arguments based on the mapping from virtual to physical disk.

4.3.3 Configuration and Deployment

While I have discussed so far how object stores are implemented and executed in the Lego framework, I next outline the two other stages involved (Figure 4.6) in a developer's use of Lego to emulate the operation of a distributed object store on a cluster of machines.

Configuration. Before deploying an object store implementation on a cluster, the developer must first instruct Lego as to how the deployment must be configured. Every configuration parameter can either be globally applicable (e.g., number of replicas maintained for every object) or apply locally to a node. To simplify the specification of local configuration parameters, Lego enables separation of nodes into roles, allowing for configuration per role. For example, distinct configurations for the master and the chunkservers in GFS; similarly, distinguishing between the directory, cache, and storage nodes in Haystack.

The format of the configuration file is straight-forward. Each configuration parameter is prefixed with the scope, then each core component or module uses a unique name (if applicable), and finally the parameter name is the suffix. Table 4.3 shows a small, simplified example of a config file.

```
//global config
global.replicationFactor = 3
global.writeConsistency = ALL
global.readConsistency = ONE
//local configs
directoryNode.metadataStore.capacity = 4GB
cacheNode.cache.capacity = 2GB
storeNode.metadataStore.capacity = 1GB
storeNode.dataStore.chunkSize = 64KB
storeNode.diskManager.volumes = "/mnt/sda,/mnt/sdb"
```

Table 4.3: Example configuration file.

Deployment. Since deploying an object store across multiple nodes with different configurations is tedious, Lego includes a *Deployment Manager* to simplify this process. To facilitate multiple node roles, the Deployment Manager allows a developer to specify a logical name and node count for every role. For example, if a developer specifies *"master=1, slaves=2"*, the Deployment Manager maps any mention of the corresponding logical hostname in the object store's implementation with physical hostnames, e.g., *"master=host26, slave1=host34, slave2=host11"*.

In addition, on start-up, the Deployment Manager invokes module-specific initialization procedures. This lets every module setup any state required to execute its stateful function. For example, a *Data Store* responsible for storing object data on disk will need to issue DISK_ALLOC requests to the Disk Manager, whereas a *Metadata Store* responsible for storing in-memory metadata will need to initialize its in-memory map. Once all modules have been initialized, the system is ready to serve requests and Lego's execution engine takes over.

4.4 Lego Implementation

Lego is written in approximately 12K lines of C++ code. Here, I present some of the implementation details not already covered in our description of Lego's design.

Lego server. I implement the Network Manager as a standard TCP server which handles Google Protocol Buffer [76] messages. The server receives incoming network messages, parses them, and inserts them into a work queue. The Event Dispatcher maintains a pool of worker threads. The workers in this thread-pool multiplex over the work queue to handle events.

To track messages corresponding to the same read/write operation as they propagate across machines, the Network Manager at any node running a client inserts an *operationID* into every PUT/GET message sent out by the client. Whenever the Event Dispatcher invokes a particular handler to process a message, it copies the operationID included in the input message into any message output by the handler. When messages are received back at the client, the Network Manager uses the operationID included in the message to decide which client thread to hand it to.

Lego client. I implement a simple, lightweight multi-threaded client, which I use across all object store implementations to issue GET and PUT requests based on an input workload. The client takes as input a list of entry nodes, port number, and a workload. The workload can be specified either as a trace file which can replayed or as a workload characterization from which Lego can generate a workload. The client measures per-operation latency, and dumps measured latencies to a local log.

	Lines of code	**Modules**	**Event Handlers**
Lego	3,628	44	43
Stand-alone (Non-Lego)	20,572	167	65

Table 4.4: Number of lines of code, unique modules, and unique event handlers in aggregate across the 10 systems implemented; comparison with Lego vs. stand-alone implementations.

4.5 Evaluation

I evaluate Lego to answer two broad questions:

- To enable rapid prototyping and evaluation of distributed object store designs, to what extent does Lego simplify the development of an object store by enabling code reuse?
- Despite imposing modularity on object store development, does Lego preserve fidelity in terms of being able to select the most cost-effective system design and hardware configuration for any particular workload?

I evaluate Lego in a variety of deployments on a cluster of 10 servers, each with two quad-core Intel Xeon 2.26 GHz CPUs, 24 GB DRAM, and eight 500 GB 7.2K RPM hard disks; all servers connect to a shared 10 GbE network.

4.5.1 Simplifying development

I evaluate the utility of Lego in simplifying the realization of distributed object store designs by using it to prototype the designs of 10 well-known systems: Cassandra, Chain replication, COPS, CRAQ, FAWN, GFS, Haystack, Memcached, Swift, and Voldemort. I rely on publicly available code, available documentation, and published research papers to faithfully mimic the design details in every system that are likely to impact performance.

Reduction in number of lines of code. First, I compare the development effort when using Lego versus when implementing every system independently. Table 4.4 compares the total

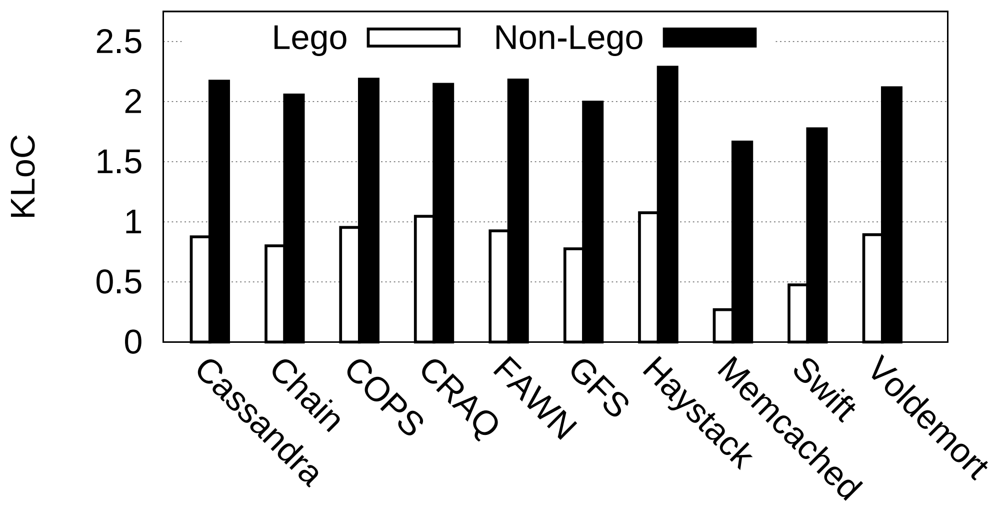


Figure 4.7: Comparison of number of lines of code between the Lego-based modular implementation and a stand-alone implementation.

number of lines of code between the two scenarios, whereas Figure 4.7 shows the comparison on a per-system basis. With Lego, every system's implementation can draw upon modules used in other systems. Additionally, the use of Lego preempts the need to implement network transport, multi-threading, managing work queues, and other core services implemented within Lego's execution engine. In total, I find that using Lego reduces the total number of lines of code to implement all 10 systems by ~17K lines of code, which corresponds to a savings of over 80%. When examining code on a per-system basis, for all 10 systems, developing the system in the manner prescribed by Lego reduces the number of lines of code by at least 2.8$\times$. Note that this comparison ignores the overhead that a developer would incur in looking up the documentation of previously implemented modules to identify which ones she can reuse; this depends on the quality of documentation and the expertise of the developer, and is hence hard to quantify.

Reuse of modules and event handlers. I dig deeper into the reuse of event handlers and modules that enables these reductions in development effort. First, I analyze the code from a per-module and per-event handler perspective. For each unique module/event handler across the

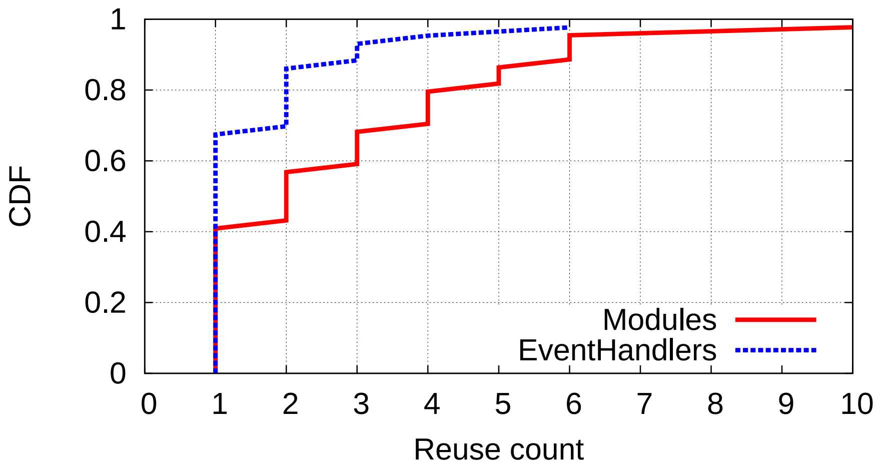


Figure 4.8: Across 10 systems implemented in Lego, the number that use each module/event handler.

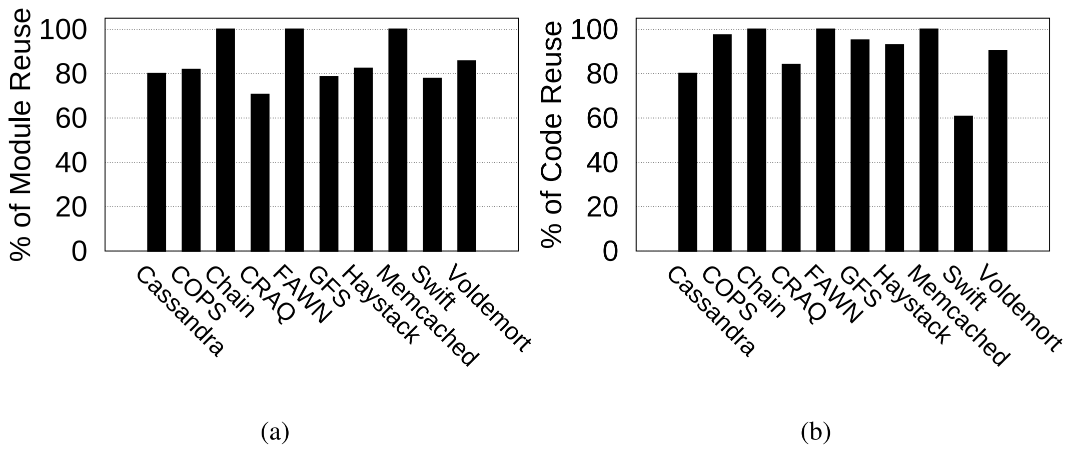
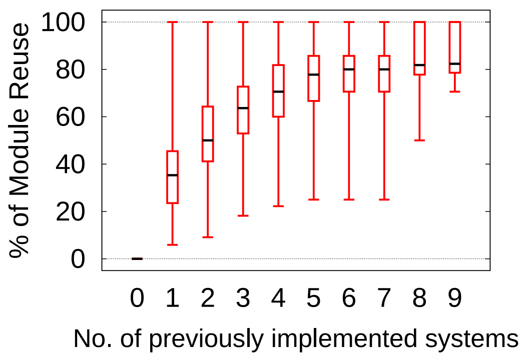


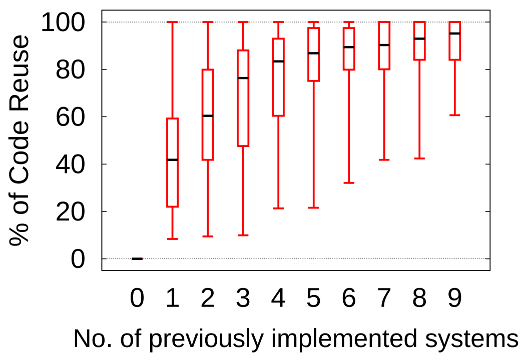
Figure 4.9: For each system, fraction of (a) reused modules and (b) reused lines of code when drawing upon implementations of other systems.

10 systems, Figure 4.8 plots the number of systems that use it. I find that 60% of modules are reused, while only 30% of event handlers are. This highlights the need for fine-grained reuse at the granularity of modules rather than at the coarser granularity of event handlers.

Next, I analyze code reuse from the perspective of each object store. In Figure 4.9, for each system, I plot the fraction of that system's modules/code that it can reuse from the other 9 systems. I see that the potential for reuse is high across all systems, both in terms of modules and lines of code; in fact, some systems can be constructed entirely by reusing modules written



(a)



(b)

Figure 4.10: Fraction of (a) reused modules and (b) reused code as a function of number of systems implemented previously; 10th, 25th, 50th, 75th, and 90th percentiles are shown across 10,000 simulated trials.

for other systems.

Impact of implementation order on reuse. The above results however assume that, when each of the systems is implemented, all other 9 systems have already been implemented. In reality, object stores are going to be implemented in Lego one after the other, and the ones implemented later (earlier) have greater (lesser) scope for code reuse.

To quantify this effect of the order in which systems are implemented, we consider 10,000 random orderings of the 10 systems. For each ordering, I compute the fraction of modules and lines of code that the i^{th} system can reuse by considering that it can only draw upon the $(i-1)$ systems implemented prior to it. For every value of i from 1 to 10, Figure 4.10 plots the fraction of reused modules and lines of code; the box plots the 25th, 50th, and 75th percentiles, and the whiskers plot the 10th and 90th percentiles. My results show that, in the median case, by the time the 3rd system is implemented, 50% of modules can be reused, and by the 6th system, over 80% of modules can be reused. When considering code reuse, over 60% of lines of code can be reused in the median case by the 3rd system and over 80% by the 5th. These results show that, by enabling module reuse, Lego can significantly reduce development effort neces-

Workload	Object Size	Read/Write Characteristics	Performance SLO
Workload1	2 MiB	Write-heavy	80 PUTs/s, 20 GET/s, 90^{th} %ile PUT latency < 20s
Workload2	4 KiB	Read-many write-many	50 PUTs/s, 50 GETs/s, 90^{th} %ile GET latency < 60ms
Workload3	64 KiB	Read-heavy	10 PUTs/s, 90 GETs/s, 90^{th} %ile GET latency < 75ms

Table 4.5: Workloads used in validation of Lego to accurately capture cost vs. performance tradeoffs.

sary to prototype new object store designs even when only a few systems have been previously implemented.

4.5.2 Enabling Evaluation of Cost-Performance Tradeoffs

The primary utility that I envision of Lego is to aid the selection of the most cost-effective object store design for any particular target setting. To evaluate Lego's ability to serve this purpose, I conduct the following case study. I choose three object stores (Swift, Voldemort, and Haystack), and for each, I consider a workload for which it was designed; Table 4.5 describes the characteristics and SLO for each workload. I then compare the cost-performance tradeoff offered by the Lego-based implementations of the three designs across the three workloads.

Latency comparison with equal cost. First, for each workload, I compare the performance offered by the three systems when they are deployed on the same cluster. In all cases, I use a replication factor of 3 and configure for strong consistency (write to all replicas, read from any replica).

Figure 4.11 compares the ability of each object store implementation to meet the specified SLOs. For each (object store, workload) pair, I plot the ratio of the GET or PUT latency SLO for that workload to the corresponding percentile of latencies measured when using that

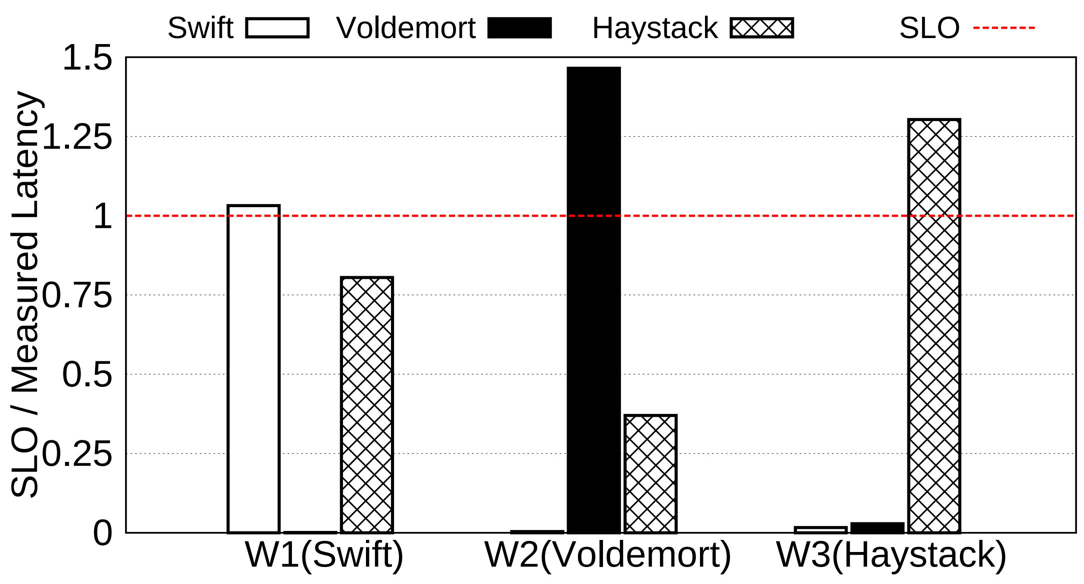


Figure 4.11: Comparison using Lego of the ability of three object store implementations to meet the SLOs for three different workloads. For each workload, the object store implementation suited for that workload meets the SLO (above red line), while the other implementations do not.

object store to serve the workload. For example, the bar for Swift on Workload1 is the ratio of 20 seconds (SLO in Workload1) to the 90^{th} percentile PUT latency when using Swift for Workload1. When this ratio is $>= 1$ (< 1), it indicates that the SLO for the corresponding workload was met (not met). I find that systems implemented on Lego match my expectations in regards to the ability of each object store to meet the SLO associated with each workload.

Cost comparison to meet SLO. Next, for each (object store, workload) pair which did not meet the workload's SLO, I scale the hardware used by that Lego deployment until the workload's SLO is met or my cluster resources are exhausted. Figure 4.12 plots the cost required to meet each workload's SLO on each object store. I only scale the number of servers or disks per server, and I consider the CPU and RAM cost as a fixed portion of the server cost. As expected, I find the cost to meet a workload's SLO on an object store which is sub-optimal for that workload is higher (12% at minimum), and can become prohibitive (4.5$\times$ or more) depending on the chosen candidate design.

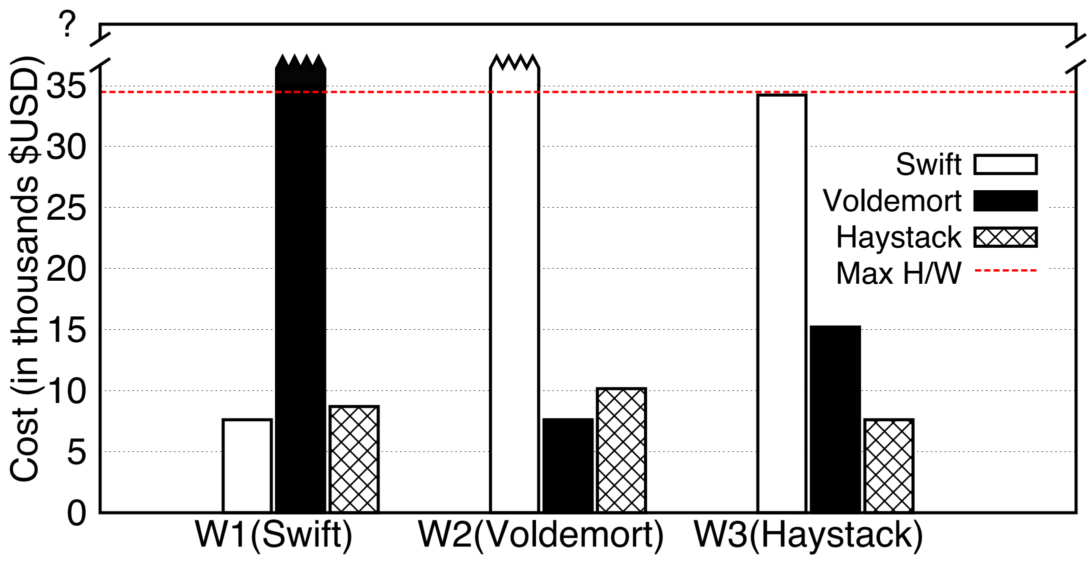


Figure 4.12: Cost required to meet each workload's SLO. The red line represents the cost when my entire cluster is used, and bars higher than the red line indicate that SLO was not met even when using the entire cluster.

Tailoring object store design to hardware. In my experiments thus far, I fixed the hardware configuration and only varied the scale. In practice, it is often not the case that the hardware setting is known apriori. There may in fact be an opportunity to co-design both the hardware configuration and the design of the object store. Therefore, a developer may want to first evaluate how different object store designs perform in different hardware settings, and only then jointly select both the object store design and hardware platform that offers the best cost-performance

	Hardware Specs.	**Object Store Design**
A	10 servers, 48GB RAM, 8 HDDs per server	Coordinated Replication, Consistent Hashing, Log Data Store, In-Memory Index Metadata Store
B	3 servers, 2GB RAM, 1 SSD + 8 HDDs per server	Coordinated Replication, Consistent Hashing, Log Data Store, On-SSD Index Metadata Store
C	3 servers, 2GB RAM, 8 HDDs per server; 1 server, 2GB RAM, 1 SSD	Coordinated Replication, Consistent Hashing, Log Data Store, On-SSD Index Metadata Store on Central Directory node

Table 4.6: Candidate object stores and hardware deployments for evaluating file backup workload.

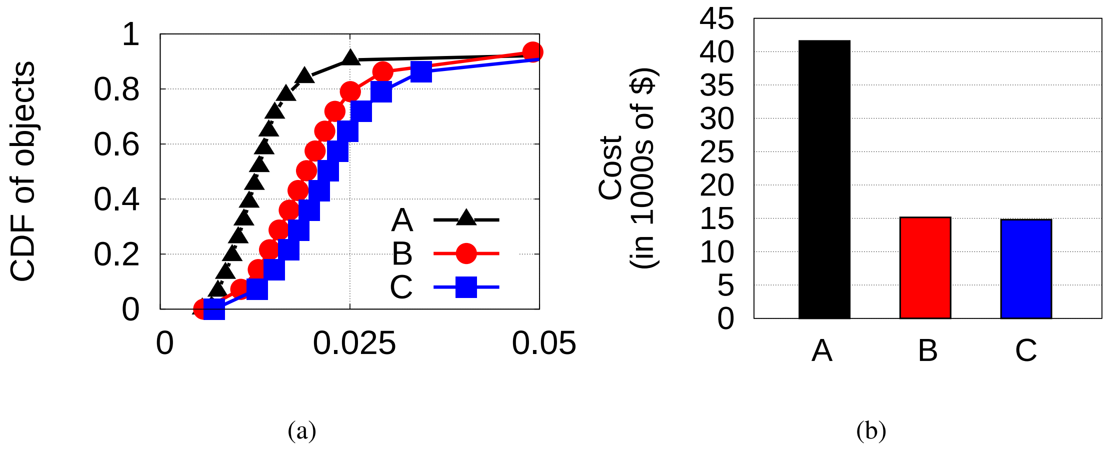
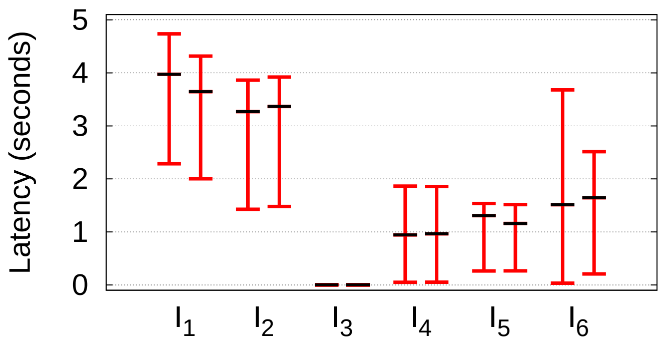


Figure 4.13: The (a) distribution of latencies, and (b) hardware cost, for three deployments when using Lego to evaluate a new file backup workload.

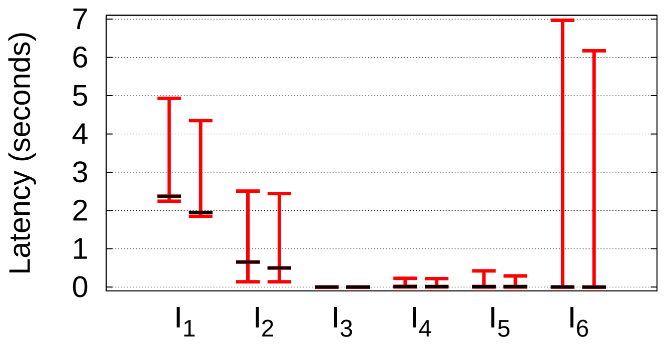
tradeoff for the target workload. I demonstrate the utility of Lego in such a scenario with the following case study.

I consider a file backup workload, which includes a read:write ratio of 1:9 on 64 KiB objects. The workload's SLO requires maintaining a throughput of 50 operations/second. I choose three candidate hardware platforms and pair each of these with a candidate object store design; Table 4.6 presents these (hardware, object store) combinations. Deployment A is designed to store all metadata in memory, while each server in Deployment B stores all metadata on a local SSD, and all servers in Deployment C offload metadata to a remote SSD at a centralized metadata store server. Each deployment is configured to split objects into 4 KiB chunks.

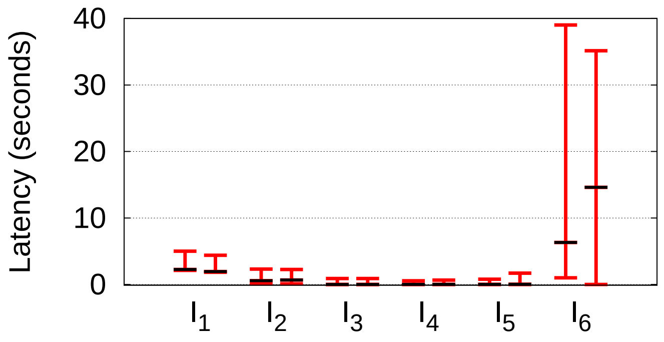
I use Lego to measure the performance offered by each of the candidate deployments and compute the hardware cost necessary to meet SLOs. Figure 4.13(a) shows the distribution of read/write latencies with each deployment, and Figure 4.13(b) shows the total cost of hardware in each case. First, Deployment A stores all metadata in memory, and due to per-server memory capacity it requires increased scale to store the metadata. I find that Deployment A offers the best performance at the highest cost. Next, Deployment B stores all metadata on a local SSD



(a)



(b)



(c)

Figure 4.14: Comparing 50th, 90th, and 99th percentile latency of three workloads, (a) update-heavy, (b) read-mostly, and (c) read-latest, between the Lego (left bar) and stand-alone non-Lego (right bar) implementations of six object stores: Chain Replication (I_1), CRAQ (I_2), Memcached (I_3), Haystack (I_4), Swift (I_5), and Voldemort (I_6).

at each server and is over-provisioned with more SSDs than required. Thus, while Deployment B offers slightly better performance than Deployment C, it does so at a higher cost. Finally, Deployment C incurs the cost of provisioning an extra server to be a centralized metadata store

Workload	Object Size	Performance SLO
Update-heavy	4 KiB	50 PUTs/s, 50 GET/s
Read-mostly	16 KiB	5 PUTs/s, 95 GETs/s
Read-latest	64 KiB	5 PUTs/s, 95 GETs/s, 5% of objs get 95% of accesses

Table 4.7: Workloads used for measuring Lego overhead.

which stores all metadata for every server on SSD storage. The performance of Deployment C is near that of Deployment B but a lower cost, so I find Deployment C is the best in terms of trading off cost and performance. This showcases the utility of Lego in enabling developers to jointly select system designs and hardware configurations based on the associated cost vs. performance tradeoffs.

4.5.3 Performance Overhead of Lego

In the final part of my evaluation, I evaluate the impact on performance when using Lego-based object store implementations compared to when every system is implemented independently without attention to modularity or code reuse.

For the purpose of measuring performance, I first implement each system on Lego, and then reimplement each system as a stand-alone non-Lego implementation which does not rely on Lego's execution engine or attempt to be modular. Rather than comparing Lego's implementations to the actual systems, we implement my own non-Lego version of each system in order to fairly compare performance results. This is because, as mentioned before, the Lego implementations aim to faithfully mimic the data path of each system, but do not implement all features (e.g., fault tolerance) of every system, some of which may have negative impact on performance.

In this experiment, I use the workloads shown in Table 4.7. We adapt these workloads from YCSB [25]. In all cases, I use a replication factor of 3 and enforce strong consistency (again, write to all replicas and read from one). I run each workload on the Lego and non-Lego variant of each object store, and compare the performance between both. Due to space limitations, I include results for only six of the ten systems I have implemented.

Figure 4.14 compares the performance by plotting the 50^{th}, 90^{th}, and 99^{th} percentile latencies across all operations (both PUTs and GETs) for both variants of each object store. For each pair of bars, the left bar is for the Lego implementation and the right bar is for the non-Lego implementation. Across the 10 systems, the average overhead in median latency is less than 10% and the overhead in 99^{th} percentile latency is below 12%. In short, my results show that the performance overhead of the Lego implementation of each system is low, and negligible in many cases.

4.6 Summary

In this chapter, I motivated the need for an emulator that can help developers select the design for an object store that can most cost-effectively satisfy their workload's performance goals. Based on the observation that different object store designs often share some similarities, I presented the design and implementation of our Lego framework that can enable rapid prototyping of system designs by facilitating module reuse. I empirically demonstrated that Lego's modularity enables significant code reuse when it is used to implement 10 popular distributed object stores, while imposing low performance overhead. I plan to release Lego to the community under an open-source license.

Chapter 5

Conclusions

In this chapter, I summarize the work I have presented and review the thesis that they support.

5.1 Thesis and Contributions

In this dissertation, I have supported the following thesis: *enabling flexibility in distributed object stores can reduce development effort when designing and implementing new applications*.

The content of this dissertation describes work completed in the development of two distributed storage systems, *Simba* and *Lego*, which, via their flexible implementations, offer improved ease of use and reduced development overhead when implementing data-centric applications.

5.1.1 Cloud-based data sync for mobile apps

In this work, I studied several popular apps and found that many exhibit undesirable behavior under concurrent use due to inadequate treatment of data consistency. Motivated by the shortcomings, I proposed a novel data abstraction, called a *Simba Table*, that unified a tabular and object data model, and allowed 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. I built Simba, a data-sync service, to demonstrate the utility and practicality of our proposed abstraction, and evaluated 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.

5.1.2 Emulation of distributed object stores

To aid the selection of the distributed object store design that can most cost-effectively satisfy performance goals, I designed and implemented Lego, an object store emulator. Lego enables developers to instantiate various system designs on a cluster of servers, compare the performance offered by them, and pick the design best suited to the target deployment scenario.

To enable rapid prototyping of any system design, I constructed Lego as a modular architecture, wherein one can instantiate any object store as a collection of modules that are oblivious to each others' implementation. Since different object store designs often share some similarities, Lego's modularity enables new system designs to be instantiated by reusing modules written for previously instantiated designs; new implementations for only one (or some) of Lego's components will be necessary.

In using Lego to instantiate 10 popular distributed object stores, I found that Lego reduced the total number of lines of code by over 80% compared to independent implementations of each system, while accurately capturing cost-performance tradeoffs across workloads and hardware configurations.

5.2 Summary

My work leverages flexible system design to improve the state of distributed storage in both enterprise and mobile environments. I examined existing distributed storage systems in order to understand how to improve their ease of use and reduce developer implementation effort through introducing flexibility. I settle on two approaches in two different settings.

First, I presented *Simba*, a cloud-based data synchronization platform which bridges-the-gap in the current status quo of mobile data synchronization platforms. Existing platforms generally offer table- or object-only synchronization, whereas Simba is designed around a novel data abstraction which unifies tabular and object data. I further improved upon current systems which only offer eventual consistency by enabling Simba to additionally offer both causal and strong consistency semantics as well.

Next, I presented *Lego*, a modular emulator for distributed object stores which reduces the burden on application developers when choosing the right distributed object store for their workload. Lego exploits the similarities across distributed object stores to offer a mix-and-match style object store development experience, and allows reuse of modules to enable rapid prototyping of new object store designs.

5.3 Future Work

In this section, I discuss directions for future research efforts related to the work I presented in this dissertation.

5.3.1 Evolvable system implementations

While I focus on a modular object store emulator, my approach for developing Lego in Chapter 4 can have broader implications. The current undesirable scenario of having to implement a new storage system from scratch for every new deployment setting stems from the fact that every system has assumptions about its target setting baked into its implementation. As a result, it is often non-trivial to modify any existing system in order to support new workload requirements or hardware characteristics. Thus, beyond simplifying the task of choosing the right design of a distributed object store, I envision that the principles underlying Lego's modular approach can also be applied to enable the development of evolvable system implementations.

Though the Lego modular emulator does not aim to reproduce a production-quality systems, evolvable system implementations which are production-quality may be useful for many applications. The deterrent from a modularly designed system is the performance overhead caused by enabling the modularity. Yet, in some cases, a modular design may be more useful than a heavily workload-optimized design. For instance, an application's workload may change over time or as the application grows. Proof of this phenomenon can be observed in the case of Facebook's photo storage. Haystack [18] originally stored all photos and optimized latency to frequently accessed "hot" photos using extensive caching. As Facebook's corpus of media objects grew, Haystack became increasingly inefficient, and f4 [63] was developed to help optimize storage of less frequently accessed "warm" data. These two systems are separate and objects must be migrated between them based on their popularity. Had Haystack been designed as a modular evolvable system, introducing storage for warm objects may have been enabled at a reduced development effort.

5.3.2 Measuring and understanding consistency trade-offs

Distributed storage systems need to define consistency semantics, or constraints on the point in time at and/or ordering in which all replica nodes will see a given update to a data object. There is a wide-breadth of flexibility when it comes to defining consistency semantics, and thus, there are many different consistency schemes and definitions across existing work. In Chapter 3, Simba provides tunable consistency to allow developers to easily enable strong, causal, and eventual consistency guarantees on a per-table basis. However, the choice of consistency scheme is still difficult to reason about even for seasoned developers. In general, strong consistency trades off predictable behavior at the cost of lower performance and reduced implementation complexity. The stronger the consistency, the lower the chance of unexpected behavior being externally visible to an application or user. Meanwhile, weaker forms of consistency (e.g., causal, eventual) increase performance at the cost of increased implementation complexity.

Understanding how the chosen consistency semantics impact the target workload (or system design) is not always straight-forward. In order to fully understand how a workload is impacted by the choice of consistency, it would be useful to measure the possible data inconsistency which could occur in the absence of a stronger consistency scheme. For example, under eventual consistency where updates are asynchronously persisted at replica nodes, determining what percentage of object reads may return out-of-date data can help a developer determine if stronger consistency scheme is necessary. Overall, the design and implementation of future distributed storage systems can benefit from measuring and understanding consistency semantics in a given setting. Such information can help to determine which is the best trade-off for a specific workload or application. A framework which helps provide such measurements would be useful for developers to analyze the performance trade-off and ultimately determine which consistency semantics to use for their data.

Bibliography

[1] Amazon simple storage service. http://aws.amazon.com/s3/.

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

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

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

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

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

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

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

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

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

[11] Windows Azure storage. http://www.windowsazure.com/en-us/home/features/storage/.

[12] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: Versatile cluster-based storage. In *FAST*, 2005.

[13] Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Mobile data sync in a blink. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, California, June 2013.

[14] Nitin Agrawal, Leo Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Emulating Goliath storage systems with David. In *FAST*, 2011.

[15] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M Hellerstein. Consistency without borders. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 23. ACM, 2013.

[16] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, 2009.

[17] Emc atmos. `http://www.emc.com/storage/atmos/atmos.htm`.

[18] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *OSDI*, 2010.

[19] Nalini Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulè, Mike Dahlin, and Robert Grimm. PADS: A policy architecture for data replication systems. In *NSDI*, 2009.

[20] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *NSDI*, volume 6, pages 5–5, 2006.

[21] Bitcask. `https://github.com/basho/bitcask`.

[22] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, June 2012.

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

[24] Byung-Gon Chun, Carlo Curino, Russell Sears, Alexander Shraer, Samuel Madden, and Raghu Ramakrishnan. Mobius: unified messaging and data serving for mobile apps. In *MobiSys '12*, 2012.

[25] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[26] Dave Engberg. Evernote Techblog: WhySQL? `http://blog.evernote.com/tech/2012/02/23/whysql/`.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[28] The DiskSim simulation environment. `http://www.pdl.cmu.edu/DiskSim/`.

[29] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, pages 481–494. ACM, 2012.

[30] Dropbox. `http://dropbox.com`.

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

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

[33] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, 1995.

[34] Facebook. `http://www.facebook.com`.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[36] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login magazine*, 38(3), June 2013.

[37] Younghwan Go, Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Reliable, consistent, and efficient data sync for mobile apps. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Santa Clara, CA, February 2015. USENIX Association.

[38] Salil Gokhale, Nitin Agrawal, Sean Noonan, and Cristian Ungureanu. KVZone and the search for a write-optimized key-value store. In *HotStorage*, 2010.

[39] Google cloud storage. `https://cloud.google.com/storage/`.

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

[41] John Linwood Griffin, Jiri Schindler, Steven W Schlosser, John S Bucy, and Gregory R Ganger. Timing-accurate storage emulation. In *FAST*, 2002.

[42] Richard G Guy, John S Heidemann, Wai-Kei Mak, Thomas W Page Jr, Gerald J Popek, Dieter Rothmeier, et al. Implementation of the ficus replicated file system. In *USENIX Summer*, pages 63–72, 1990.

[43] Shuai Hao, Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Building a Delay-Tolerant Cloud for Mobile Data. In *Proceedings of the IEEE MDM Conference*, June 2013.

[44] Danny Harnik, Ronen Kat, Oded Margalit, Dmitry Sotnikov, and Avishay Traeger. To Zip or not to Zip: Effective Resource Usage for Real-Time Compression. In *FAST 2013*, Feb 2013.

[45] HDFS: Hadoop distributed file system. `http://hadoop.apache.org`.

[46] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. In *VLDB*, 2011.

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

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

[49] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[50] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks are like snowflakes: No two are alike. In *HotOS*, 2011.

[51] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[52] Butler W. Lampson. Hints for Computer System Design. In *SOSP '83*, pages 33–48, Bretton Woods, NH, October 1983.

[53] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets*, 2010.

[54] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Pregui, Rodrigo Rodrigues, Alexander Wieder, Parmod Bhatotia, Ansley Post, Rodrigo Rodrigues, et al. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation*, pages 175–188. USENIX, 2012.

[55] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[56] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. A short primer on causal consistency. *USENIX ;login magazine*, 38(4), August 2013.

[57] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Symposium on Networked Systems Design and Implementation*, 2013.

[58] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

[59] Harsha V. Madhyastha, John C. McCullough, George Porter, Rishi Kapoor, Stefan Savage, Alex C. Snoeren, and Amin Vahdat. scc: Cluster storage provisioning informed by application characteristics and slas. In *FAST*, 2012.

[60] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with Anvil. In *SOSP*, 2009.

[61] Jeffrey C. Mogul. The case for persistent-connection http. pages 299–313, 1995.

[62] Robert Morris, Eddie Kohler, John Jannotti, and M Frans Kaashoek. The Click modular router. In *SOSP*, 1999.

[63] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm blob storage system. In *OSDI*, 2014.

[64] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-Bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 174–187, Lake Louise, Alberta, October 2001.

[65] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

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

[67] Netty project. `http://netty.io`.

[68] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 276–287, New York, NY, USA, 1997. ACM.

[69] ns-3 network simulator. `http://www.nsnam.org/`.

[70] OpenStack Swift. `http://swift.openstack.org`.

[71] Alina Oprea and Michael K. Reiter. On consistency of encrypted files. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 254–268, Berlin, Heidelberg, 2006. Springer-Verlag.

[72] Douglas Stott Parker Jr, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, (3):240–247, 1983.

[73] Parse. `http://parse.com`.

[74] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST*, 2008.

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

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

[77] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *Parallel and Distributed Systems, IEEE Transactions on*, 14(8):730–741, 2003.

[78] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: a platform for content-based partial replication. In *NSDI'09*.

[79] Kai Ren and Garth Gibson. TABLEFS: Embedding a NoSQL database inside the local file system. In *APMRC*, November 2012.

[80] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[81] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[82] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, , Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *FAST '13*, February 2013.

[83] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[84] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-grained data management abstractions for modern operating systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 113–129, Broomfield, CO, October 2014. USENIX Association.

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

[86] Rachel Swaby. With Sync Solved, Dropbox Squares Off With AppleâĂŹs iCloud. `http://tinyurl.com/dropbox-cr`, December 2011.

[87] OpenStack Swift. `http://swift.openstack.org`.

[88] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *ATC*, 2009.

[89] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP '95*, New York, NY, USA, 1995.

[90] Douglas B Terry. *Replicated Data Management for Mobile Computing*, volume 5. Morgan & Claypool Publishers, 2008.

[91] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

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

[93] Niraj Tolia, Mahadev Satyanarayanan, and Adam 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.

[94] Twitter. `http://www.twitter.com`.

[95] Peter Urban, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *IEEE ICOIN*, 2001.

[96] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI*, 2002.

[97] Robbert van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. Efficient Reconciliation and Flow Control for Anti-entropy Protocols. In *LADIS '08*, pages 6:1–6:7, New York, NY, USA, 2008. ACM.

[98] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[99] Project Voldemort. `project-voldemort.com`.

[100] Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering researchers to evaluate large-scale storage systems. In *NSDI*, 2014.

[101] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzah. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[102] Brian White, Shashi Guruprasad, Mac Newbold, Jay Lepreau, Leigh Stoller, Robert Ricci, Chad Barb, Mike Hibler, and Abhijeet Joglekar. Netbed: an integrated experimental environment. In *OSDI*, 2002.

[103] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[104] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sergio Duarte, Carlos Baquero, Marc Shapiro, and Nuno M. Preguica. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *CoRR*, pages –1–1, 2013.

[105] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, Broomfield, CO, October 2014. USENIX Association.

[106] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.