©Copyright 2020
Niel Lebeck

Simplifying High-Performance Data Management for Distributed Applications

Niel Lebeck

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Arvind Krishnamurthy, Chair

Henry M. Levy, Chair

Irene Zhang

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Simplifying High-Performance Data Management for Distributed Applications

Niel Lebeck

Co-Chairs of the Supervisory Committee:
Professor Arvind Krishnamurthy
Paul G. Allen School of Computer Science & Engineering

Professor and Wissner-Slivka Chair Henry M. Levy
Paul G. Allen School of Computer Science & Engineering

Modern distributed apps aim to provide high-fidelity experiences, with natural user interfaces and seamless interaction between users around the world. However, distributed apps face fundamental constraints on network latencies and mobile device computational power. These constraints make life hard for app developers, requiring them to use high-latency resources such as the disk and network. When working with high-latency resources today, developers must use cumbersome asynchronous programming models, keep app data stored in multiple locations with different representations, and reason about tradeoffs in the locations of data and computation. This thesis presents three systems that simplify distributed app development by providing developer-friendly interfaces to high-latency resources. *Diamond* allows apps to access shared data with strong guarantees and scalable performance. *Hercules* lets apps expose uncertainty in shared data and respond instantly to user input. *Marvin* provides a transparent interface for swapping memory to disk that is tailored to mobile devices. We show that these systems simplify the developer experience, experimentally demonstrate that they do not sacrifice performance, and suggest directions for future work.

TABLE OF CONTENTS

Page

List of Figures v

Chapter 1: Introduction 1
- 1.1 Types of Distributed Applications 3
- 1.2 The Architecture of Distributed Applications 4
- 1.3 Systems for Distributed Applications 6
  - 1.3.1 Backend Components 6
  - 1.3.2 Mobile Clients 8
  - 1.3.3 Cross-Tier Backend Systems 10
  - 1.3.4 Cross-Layer Backend Systems 10
  - 1.3.5 Systems Spanning Client and Cloud 10
- 1.4 Thesis Systems 12

Chapter 2: Diamond 16
- 2.1 Introduction 16
- 2.2 Diamond Overview 17
  - 2.2.1 System Model 17
  - 2.2.2 Reactive Data Types 17
  - 2.2.3 Transactions 18
  - 2.2.4 Diamond Architecture 19
  - 2.2.5 Transaction Protocol 21
- 2.3 Reactive Transactions 22
  - 2.3.1 Interface 24
  - 2.3.2 Protocols 25
  - 2.3.3 Data Push Notifications 27
- 2.4 Comparison with BaaS Systems 27

2.4.1 The Difficulty of Designing a BaaS Interface 28
2.4.2 Application Operations 30
2.4.3 Reactive Updates 37
2.5 Implementation 44
2.6 Evaluation 44
2.6.1 Programming Experience 44
2.6.2 Performance Evaluation 46
2.7 Conclusion 51

Chapter 3: Hercules 52
3.1 Introduction 52
3.2 Motivation 55
3.2.1 The Needs of Real-Time Interactive Apps 55
3.2.2 Current Systems 58
3.3 Hercules Overview 59
3.3.1 System Model 60
3.3.2 Programming Interface 60
3.3.3 Hercules State Views 61
3.4 Hercules Design 63
3.4.1 AMBROSIA Background 63
3.4.2 System Architecture 64
3.4.3 Operation RPC Protocol 66
3.4.4 Client View Computation 67
3.4.5 Client Rebase Batching 70
3.4.6 Recovery from Failures 71
3.5 Implementation 73
3.6 Application Case Studies 74
3.6.1 Interior Design App 74
3.6.2 Chat App 75
3.6.3 Spreadsheet App 76
3.7 Evaluation 81
3.7.1 Setup 81
3.7.2 Baseline Performance 82

3.7.3 Effect of Client Rebase Batching . . . 82
3.7.4 Recovery from Client Failures . . . 84
3.8 Related Work . . . 84
3.9 Conclusion . . . 87

Chapter 4: Marvin . . . 88
4.1 Introduction . . . 88
4.2 Limitations of Modern Mobile OS Memory Resource Management . . . 90
4.2.1 Fixed Memory Allocation . . . 91
4.2.2 No Memory Overcommit . . . 93
4.3 Our Approach . . . 94
4.3.1 Object-Level Working Set Estimation . . . 95
4.3.2 Ahead-of-time Swap . . . 96
4.3.3 Bookmarking Garbage Collector . . . 96
4.4 Marvin Overview . . . 97
4.4.1 Design Goals . . . 97
4.4.2 Marvin System Model . . . 97
4.4.3 Marvin Architecture . . . 98
4.4.4 Marvin Memory Management Timeline . . . 99
4.5 Marvin Core Mechanisms . . . 100
4.5.1 Stubs for Object Reference Indirection . . . 100
4.5.2 Reclamation Table for OS-Runtime Coordination . . . 101
4.5.3 Object Access Interposition . . . 101
4.6 Marvin Memory Management . . . 102
4.6.1 Working Set Estimation . . . 102
4.6.2 Ahead-of-Time Swapping . . . 103
4.6.3 Bookmarking Garbage Collector . . . 105
4.6.4 Design Tradeoffs and Alternatives . . . 106
4.7 Marvin Prototype . . . 107
4.7.1 Object Access Interposition . . . 108
4.7.2 Potential Optimizations . . . 109
4.8 Evaluation . . . 110

4.8.1 Evaluation Setup 110
4.8.2 Memory Reclamation 111
4.8.3 Memory Utilization 113
4.8.4 Runtime Overhead 114
4.9 Related Work 118
4.10 Conclusion 119

Chapter 5: Conclusion 121
5.1 Future Work 121

Bibliography 124

LIST OF FIGURES

Figure Number Page

1.1 Distributed application architecture. 5

1.2 The thesis systems' scope across the tiers of a distributed app and the layers of the software stack. 15

2.1 Reactive data types. 18

2.2 Diamond's isolation levels, in order of decreasing guarantees. Each isolation level for read-write transactions corresponds to the adjacent level for reactive transactions. 19

2.3 **Diamond architecture.** Shaded areas represent Diamond components. . . 20

2.4 **Diamond read-write transaction and reactive transaction protocols.** In this example, Alice runs a read-write transaction that reads record A and writes to record B, while Bob registers a reactive-transaction that reads record B. 23

2.5 The DOCC validation matrix, specifying whether a transaction with the given operation can commit (C) or must abort (A) in the presence of a conflicting prepared operation. Each prepared operation column is divided into results for the different isolation levels: Read Committed (RC), Snapshot Isolation (SI), and Strict Serializability (SS). 23

2.6 In distributed apps built with the three-tier architecture (a), clients send coarse-grained messages to the middle tier, whereas in apps built with BaaS systems (b), clients directly read and write shared data. 29

2.7 Peak throughput of Diamond compared to our baseline Redis/Jetty implementation. Diamond's isolation levels are Read Committed (RC), Snapshot Isolation (SI), and Strict Serializability (SS). 47

2.8 Throughput improvement by transaction type with DOCC. 49

2.9 Round latencies for our 100 game benchmark with and without data push notifications. 50

3.1 A sequence of screenshots from our interior design app, showing how it exposes uncertainty to the user. In (b), the user has just moved one of the chairs, and that change has not yet been synchronized with the cloud or other users, so the client highlights the chair in red. 56

3.2 An illustration of the relationship between staleness and consistency. A highly available state view cannot be both up-to-date and strongly consistent, so each state view represents a tradeoff between the two properties. 57

3.3 An example of the relationship between a client's state views and Hercules operation logs. Client A's operations A1 and A2 are in the global log, but only A1 is visible to client B. Its operation A3 is guaranteed to eventually make it into the global log, but no such guarantee applies to A4. 62

3.4 The architecture of a Hercules application. Shaded areas represent components provided by Hercules. 65

3.5 The Hercules operation protocol. Dashed lines represent the client app reading updated state views from the client library by calling the `GetState()` method. 67

3.6 An example of the correspondence between (a) the progress of operations through Hercules, (b) the operation logs reflected in the client library's state views, and (c) its operation lists used to compute those state views. 68

3.7 The Hercules client and server RPC interfaces. 69

3.8 The interior design app's output-rendering code. 75

3.9 A screenshot of the chat app client, showing the different possible message statuses. 76

3.10 The chat app's output-rendering code. 77

3.11 A sequence of screenshots from the spreadsheet app client as two users edit adjacent cells. In (b), the user has entered a value in the "Qty" column and moved the cursor to the "Price" column, but those actions are not yet visible to other users. (Colors edited for better greyscale contrast.) 78

3.12 The spreadsheet app's output-rendering code. 80

3.13 The average delay before a submitted operation in the spreadsheet app is reflected in the client's state views for both the Hercules and Cloud Firestore versions. 83

3.14 **Effect of batching as the number of clients increases.** Rebase batching allows clients to cope with the added load introduced by more clients. 84

3.15 **Effect of client failure on view staleness.** When one client fails, the measured client's *Authoritative* view is unaffected, but its operations are delayed from being applied to its *Visible* view until the visibility set timeout kicks in. 85

4.1 CDF of object size and heap percentage occupied by objects that size or smaller. Popular Android apps have a bimodal distribution where most objects are either significantly smaller or larger than a 4KB page. 91

4.2 **The cost of fixed allocation.** Each bar shows the total Java heap size of a popular app alongside its minimum Java working set during active use. While apps have large memory footprints, they do not use most of that memory, which could be better utilized for running another app. 92

4.3 **Progress over time of a memory allocation on Android with and without swap.** Android allocates all 512MB of memory in 450 ms when memory is free, while swap increases that time to almost 8 seconds. 94

4.4 **The cost of re-starting apps.** Modern mobile OSes kill apps when memory runs out rather than swapping to disk. This wastes significant time for popular apps, which take anywhere from 4x to 27x longer to restart than to read *the entire app memory image from disk*. 95

4.5 A timeline of actions performed by Marvin's swap mechanism as compared to traditional (e.g., Linux) swap mechanisms. Events are listed above the timeline while Marvin's actions in response are listed below. 99

4.6 Memory usage as Marvin reclaims memory from a benchmark app with a 500MB heap and different working set sizes. Marvin took 108ms to reclaim 500MB, much faster than the nearly 8 seconds required by Android with Linux swap to allocate the same amount of memory. 111

4.7 Count of active apps over time when starting instances of our benchmark app. Marvin runs more than twice as many apps as regular Android before needing to kill any apps; on Android with a swap file, most apps are alive but inactive due to constant swapping activity. 112

4.8 PCMark for Android benchmark results. Marvin's score is within 15% of Android's for both benchmarks, showing that Marvin adds low overhead for accessing non-reclaimable objects for real-world apps. 115

4.9 Overhead of Marvin for a synthetic workload with different proportions of DEX instructions with object access interposition (OAI). The point (0,0) represents the theoretical scenario of running no DEX instructions with OAI, while the other points show experimental results for the given workload mix. Marvin's overhead is lower when a smaller fraction of instructions have OAI. 115

4.10 Speed of a benchmark app as it touches objects in its heap with different fractions of reclaimed objects. Object faulting never occurs in the user-visible foreground app due to Marvin's policies, but when Marvin does need to fault objects in from disk on-demand, its speed matches the expected result of trading off memory accesses for disk reads. 117

ACKNOWLEDGMENTS

I'm grateful to many people for making my time at the University of Washington such an enjoyable experience. First and foremost, my advisors made me into the researcher I am today. Irene Zhang has been an invaluable source of hands-on mentorship on how to build systems and write papers, and Arvind Krishnamurthy and Hank Levy gave me big-picture guidance and the freedom to follow my interests. I'm also thankful to Dieter Fox for advising me during my first year at UW as a robotics student, and to Pankaj K. Agarwal and Thomas Mølhave for mentoring me as an undergraduate researcher at Duke University.

I've also been fortunate to work with excellent collaborators. Jonathan Goldstein at Microsoft Research mentored me during a summer internship where I had a great time and started one of my favorite research projects. Alec Wolman, Stefan Saroiu, and Sharad Agarwal mentored me during an earlier internship, where I got valuable software engineering experience. Brandon Holt, Raymond Cheng, and Pedro Fonseca were collaborators on my first project in the Systems Lab, and I learned a lot about systems research from working with them.

I owe a great deal to the staff in the Paul G. Allen School. Elise Dorough and the graduate program advising staff have been a great source of advice and support throughout my time in the PhD program. Melody Kadenko always helped me figure out how to get the equipment and supplies I needed for experiments, and the CSE Support staff helped me out whenever I ran into technical issues. Tracy Erbeck and the facilities team made the CSE buildings great places to work, and Sophie Ostlund and the operations team made CSE parties and events something I looked forward to every year.

The coffee shops around UW became a second home for me towards the end of my PhD

and the place where I did some of my most productive work. I'm grateful to Cafe Solstice—the only place that knew my order as soon as I walked in—as well as Cafe Allegro and the Henry Art Gallery cafe for giving me places to write and think.

I've had a great time in the Systems Lab due in large part to the excellent group of students and postdocs in the lab. Adriana Szekeres, Jialin Li, Naveen Kr. Sharma, Danyang Zhou, and Antoine Kaufmann welcomed me into the lab and made me feel at home as a first-year graduate student who didnt know very much about systems research at all. It's been a joy to work alongside the current lab members—Kevin Zhao, Lequn Chen, Ashlie Martinez, Samantha Miller, Henry Schuh, Jialin Li, Tianyi Cui, Pratyush Patel, Katie Lim, Tapan Chugh, Anna Kornfeld Simpson, Ming Liu, Ellis Michael, Kaiyuan Zhang, Yuchen Jin, Helgi Sigurbjarnarson, and Luke Nelson—and I can't wait to see what the Systems Lab accomplishes in the future.

Finally, I couldn't have made it through my PhD without the support of my family. My parents, Alvin and Mitali Lebeck, have been a constant source of love and encouragement. My twin brother, Kiron, was always eager to share in my excitement or sympathize during the tough times. Attending the same PhD program as him was an unexpected joy.

DEDICATION

To my family: my mother, Mitali; my father, Alvin; and my twin brother, Kiron.

Chapter 1

INTRODUCTION

Over the past several decades, as computing systems have dramatically expanded in power, scale, and reach, the devices and interfaces that people use to interact with these systems have changed dramatically as well. Mainframe terminals gave way to personal computers with mice and graphical user interfaces, and today, smartphones and tablets are replacing personal computers for a new generation of technology users. In time, new technologies such as 3D headsets may become similarly ubiquitous.

One overall trend is towards *natural user interfaces*: interfaces that act as an extension of users' senses. These interfaces convert high-fidelity, familiar human actions (e.g., touch, gesture, and voice) into application input, and they provide application output as high-fidelity sensory experiences (e.g., high-resolution video, haptic feedback, or surround-sound audio). Mobile devices such as smartphones, which accept touch input and render high-resolution video output, are a popular example of natural user interfaces today.

Another trend is towards *distributed applications*, apps that run and communicate across many devices to create a shared experience for multiple users. Users interact with distributed apps through client applications running on smartphones, tablets, or personal computers. Those client apps, in turn, connect via the internet to applications and services running "in the cloud," on servers in datacenters around the world.

An important property of devices with natural user interfaces is that they must respond quickly to user input. When humans interact with the real world, that interaction happens with zero latency: you push on a coffee cup, and you instantly see the cup move. Natural user interfaces mimic real-world sensory experiences, so they must also have low response times in order to be convincing. The precise bounds are dictated by human perception and

psychology, and have been studied by human-computer interaction researchers over the years. Miller [73] gives 100-200ms as the maximum delay for responding to "control activation" such as pressing a key or moving a switch. Nielsen [80] concurs, phrasing this limit as a 100ms maximum response time in order to give users the feeling "that they are directly manipulating objects in the UI." Higher-fidelity interfaces have even tighter bounds: Abrash [2] says that modern virtual reality headsets have only 7-20ms to update their output in response to user head movement. The applications running on these devices are ultimately responsible for ensuring that their interfaces respond quickly to user input. They must quickly execute code that lies on the critical path for processing input and rendering output. In particular, code on the UI critical path cannot block and wait for long-running synchronous operations to complete.

This requirement conflicts with the fact that modern distributed applications increasingly depend on high-latency resources for data management. Distributed app clients must communicate with far-away cloud servers or other users' clients to coordinate shared data accesses, over networks where a single round trip can take hundreds of milliseconds. Today's distributed apps also use large amounts of data in rich formats, such as images, videos, or even machine learning models. They cannot fit all of this data into main memory at the same time, so they store the unused data on disk, but moving data between disk and memory can require tens of milliseconds. Given the high latencies of the network and disk, apps cannot synchronously access either resource without violating their responsiveness requirements.

As a result, developers today must overcome a number of hurdles to effectively manage data in their distributed apps. They must use cumbersome asynchronous programming models, they must reason about the scheduling of their code across threads, and they must keep app data stored in multiple locations with different representations. Libraries exist to help developers manage disk and network interaction, but they are specific to particular types of data (e.g., image caching libraries [95, 13]). General-purpose systems and frameworks that help developers manage disk and network accesses are a sorely needed niche in the ecosystem of mobile app development.

This thesis presents systems that simplify data management by providing developer-friendly interfaces for distributed applications to interact with the disk and network. *Diamond* enables wide-area distributed apps to manage shared data using simple synchronous interfaces that provide strong guarantees. *Hercules* continues this lineage of helping distributed apps manage shared data; it provides an interface for real-time interactive apps to access shared state that explicitly exposes uncertainty in the shared state. *Marvin* provides a transparent interface for swapping unused memory to disk that is tailored to the needs of mobile devices. These systems are united by their focus on exposing interfaces that are easy to use and that provide clear and strong guarantees.

The remainder of this chapter provides background on the nature and architecture of distributed applications, surveys related work from the distributed systems and operating systems communities, and describes how this thesis's systems fit into that context.

1.1 Types of Distributed Applications

While there is considerable diversity among today's distributed apps, two broad classes of apps emerge. One class consists of large-scale distributed apps that connect millions of users in one experience. These apps typically take seconds or minutes to propagate updates throughout the system and impose user hierarchies that control the flow of updates. One example is a social media service such as Twitter, which lets users post short messages ("tweets") and subscribe to messages from ("follow") other users. Some Twitter users are celebrities with millions of followers, and delivering their tweets to all of their followers can take several minutes [53]. Another example is a news app such as the New York Times, where the news organization publishes articles that users read and comment on.

Another class consists of *real-time interactive apps*, which let small groups of users manipulate a set of shared data and interact with each other in real time. This class includes collaborative drawing and design apps, multiplayer games, messaging apps, and collaborative office apps like Google Docs. In these apps, the number of users participating in the same shared experience and the shared data they manipulate are both much smaller than in

the large-scale apps mentioned above; however, these apps must propagate updates between users in milliseconds rather than minutes.

1.2 The Architecture of Distributed Applications

Distributed applications exist as a collection of application instances running on devices, or *nodes*, ranging from end-user client devices such as smartphones to servers in datacenters. Each node runs a software stack that extends from its underlying device hardware to the application. We outline a typical distributed application's architecture by first considering the relationship between nodes and then examining the software stack on an individual node.

The three-tier architecture. A common way of organizing the nodes in a distributed application is the *three-tier architecture*, shown in the top half of Figure 1.1. This architecture separates the app into three tiers: clients, application servers, and persistent storage. Clients run on end-user devices, accepting user input and displaying output. The persistent storage tier, consisting of databases or storage systems running on cloud servers, is responsible for storing, updating, and responding to queries over an app's ground-truth shared state. The middle tier of application servers mediates between clients and persistent storage, turning client requests into storage operations. Clients are separated from the middle tier by wide-area network connections. The middle tier and storage tier are both located in cloud datacenters, so they typically communicate over low-latency and high-bandwidth connections.

The middle tier plays an important role in ensuring that a distributed app is performant and scalable. By converting a single client request into multiple storage operations, it reduces the number of wide-area round trips required to execute the client request. Its application servers are typically stateless, allowing the tier to elastically increase or decrease the number of servers and spread client requests across different servers to adapt to changing client load. Furthermore, the middle tier often includes caches that reduce load on the storage tier. The middle tier also serves an important security function, authenticating clients and preventing

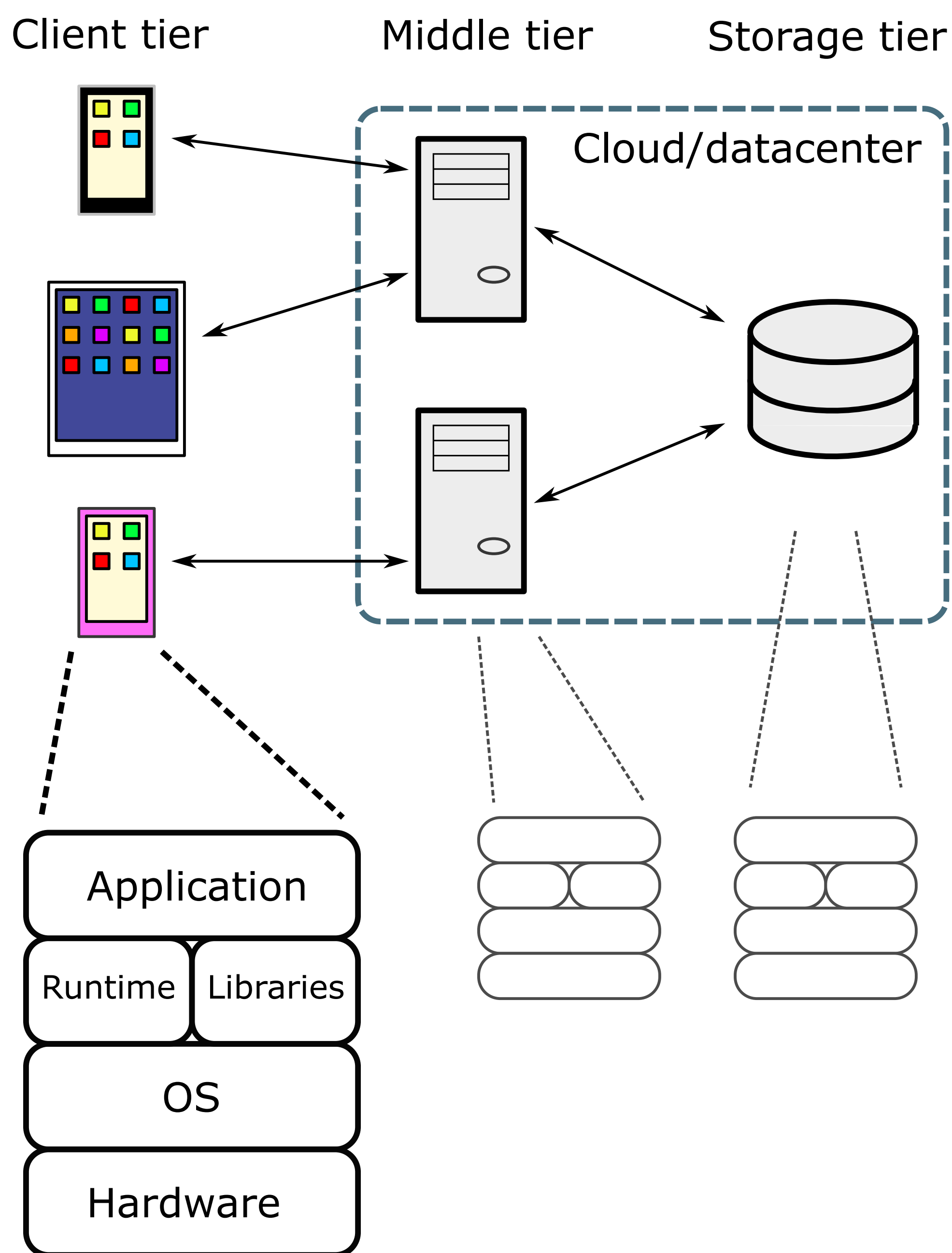

Figure 1.1: Distributed application architecture.

them from directly accessing the app's private or sensitive data.

Layers within a node. A node in each tier of the three-tier architecture consists of several software layers, shown in the bottom half of Figure 1.1. At the bottom, each node's operating system (OS) manages its hardware. The OS multiplexes limited hardware resources across multiple applications and isolates those applications from each other. Between the OS and an application, libraries run as part of the application but encapsulate common functionality into reusable building blocks. Applications written in many high-level programming languages run on top of language runtimes, which include core libraries for the language, services that transparently provide functionality such as garbage collection and thread scheduling, and interpreters or compilers for application code.

1.3 Systems for Distributed Applications

Research in the distributed systems and operating systems communities has produced a vast array of systems that support distributed applications. Some of these systems target specific tiers of the three-tier architecture or particular layers of a node's software stack, while other systems innovate by spanning multiple tiers or layers. This section surveys this rich body of work in order to illustrate its diversity and place this thesis's systems in context.

1.3.1 Backend Components

Many systems implement a specific component of the three-tier architecture's cloud or backend portion. Research focuses on improving these components' performance as their workload scales and refining their interfaces to be general and easy-to-use for developers.

Distributed storage. The storage tier of a distributed application plays a crucial role: it maintains and mediates access to the application's underlying data. It ideally provides a straightforward interface for manipulating data, is fault-tolerant when failures occur, and delivers good performance that scales with increasing numbers of users. Many storage systems

use a transactional programming model, in which the developer specifies transactions that execute as isolated and atomic units, to satisfy the first goal. The latter two goals require distributing the storage system across multiple nodes. Storing multiple copies of each data item on different *replicas* provides fault-tolerance, and partitioning data items across *shards* provides scalable performance.

A storage system with a transactional programming model requires a *concurrency control protocol* to ensure that transactions on a single node execute as isolated and atomic units. Two-phase locking [32] and optimistic concurrency control (OCC) [60] are two standard concurrency control protocols. Distributed storage systems also require an *atomic commitment protocol* to ensure that shards decide whether to execute transactions as a group, either all committing or all aborting a given transaction. Two-phase commit [49] is the canonical atomic commitment protocol.

There is also the question of how to synchronize the replicas within one shard and ensure that the system can handle replica failures. Consensus protocols, such as Paxos [61] and its variants [74, 62] or RAFT [84], and closely-related replicated state machine protocols, like Viewstamped Replication [83], are a key building block. Consensus protocols provide a mechanism for nodes to agree on a single value that is robust to failures; replicated state machine protocols similarly provide a way for nodes to agree on the contents of a state machine's operation log.

For many years, conventional wisdom held that consensus protocols were too slow to use for actually replicating a storage system's data. Instead, system designers used consensus protocols to build heavyweight coordination services such as Chubby [15] and ZooKeeper [54]. Storage systems like GFS [38] then used those coordination services to implement simple primary-backup replication schemes, in which a single "primary" or "leader" node serializes a given replica group's updates. The expensive coordination service must be invoked only when a leader node fails and a new leader must be chosen.

Spanner [20] is a recent geo-distributed storage system that bucks this trend, using Paxos to replicate each shard's data in its entirety. Spanner provides strong consistency with a

twist: it defines a *TrueTime* interface that provides timestamps with bounds on clock skew, and it uses TrueTime to optimize performance of read-only transactions. The TrueTime API returns a time interval rather than a single timestamp when queried, and it guarantees that the returned interval will contain the actual time. Spanner uses the TrueTime API to perform read-only transactions without locking, which improves that workload's performance when running Spanner with wide-area latency gaps between nodes.

Dynamo [22], in contrast, only provides single-key read and write operations instead of transactions, and it only guarantees eventual consistency: the completion of a write does not guarantee that all subsequent reads will reflect it, and clients must manually reconcile divergent update histories. Dynamo's designers found that there were many services within Amazon that needed a storage system with low tail latencies and high availability, and that could "tolerate ... inconsistencies" in order to receive those benefits.

Caching and notifications. The middle tier also contains important distributed application components. Caches such as memcached [28] reduce the pressure on the storage tier by handling the most common or frequent requests, preserving performance as a distributed application scales to include more users and data. Notification services like Thialfi [3] alert clients when relevant shared data items change or events occur. Some systems are general-purpose components that fit into multiple tiers. Redis [91] is an in-memory key-value store that is typically used as a middle-tier cache, but its persistence and replication features mean that it could be used in the storage tier as well.

1.3.2 Mobile Clients

Modern distributed applications provide clients for a range of devices and platforms, and mobile devices have emerged as a particularly important class of platforms to support. Smartphones and tablets are ubiquitous in everyday life, and the Android mobile operating system is now the most popular operating system in the world [71].

A large burst of mobile systems research accompanied the introduction of the open-source

Android OS in the late 2000s. The initial excitement around mobile systems has cooled off to an extent, but researchers continue to develop new systems targeting mobile platforms. Some of these systems explicitly consider mobile devices as client platforms in larger distributed applications, while other systems limit their focus to the mobile device itself, remaining agnostic to the question of whether a mobile app is a standalone application or a distributed application client.

Memory management. A body of recent mobile systems work focuses on improving memory management. Classic Linux swap is disabled on Android, and the OS deals with memory pressure by simply killing applications when memory runs out. These systems seek to re-introduce swapping and improve its performance. SmartSwap [112] predicts which apps are unlikely to be used and swaps out pages from those apps ahead-of-time. A2S [58] takes the opposite approach; it avoids swapping out pages from unused apps, since their pages will be freed anyways when they are terminated. MARS [51] optimizes Linux swapping to improve performance on flash storage devices. It disables garbage collection in background apps and reclaims memory from those apps. DR. Swap [111] proposes using NVRAM rather than flash storage to store swapped-out pages and satisfying reads by reading directly from NVRAM. Choi et al. [17] improve the performance of an in-memory file system by co-designing the swap mechanism to minimize I/O.

Security. Other systems modify mobile platforms to track how applications use and propagate data. These systems limit their analysis to applications running on the mobile device, but by tracking which data is downloaded or uploaded, they provide information about the security properties of distributed applications. TaintDroid [30] and TaintART [98] modify the Dalvik interpreter and the ART compiler, respectively, to add information-flow tracking; SandTrap [88] uses binary instrumentation to support information-flow tracking in native libraries across multiple threads. CleanOS [99] builds on TaintDroid to minimize the exposure of sensitive data by encrypting the data and evicting the on-device copy of the encryption

key when the data is not in use.

1.3.3 Cross-Tier Backend Systems

Several recent systems limit their scope to the cloud portion of a distributed application but span multiple tiers within the cloud, achieving performance or usability gains by extending across tiers. One such line of work concerns auto-sharding systems that control data placement across shards in the underlying storage system. Slicer [4] makes sharding decisions by itself and leaves applications to handle data migration, while Akkio [5] requires applications to partition data and takes care of data migration. Caches also benefit from integrating with the underlying storage system; TxCache [87] is a cache that provides strong consistency properties in part by requiring the storage system to expose versioning information.

1.3.4 Cross-Layer Backend Systems

While the distributed systems above generally limit their scope to the applications running on each node and the libraries used by those applications, some recent distributed systems extend lower in the software stack, using hardware or OS modifications for performance improvements. One such line of work focuses on building high-performance, low-level messaging and data manipulation primitives. FaRM [29] uses the RDMA capabilities of modern NICs to unify the memory of a cluster's machines into a single-address-space, transactional data store. eRPC [56] instead provides an RPC system that is designed with the hardware properties of modern NICs in mind but that uses ordinary packet-based networking instead of RDMA.

1.3.5 Systems Spanning Client and Cloud

Our final category includes systems that extend across the entire three-tier architecture, spanning from the end-user client application into the backend. This category includes systems aimed at simplifying application development, conserving energy on client devices,

and improving the end-user experience.

Backends-as-a-service. Some systems abstract away the middle and storage tiers entirely, encapsulating them in an opaque "cloud" that presents a simple interface for clients to interact with shared data. Recent systems in this category have branded themselves as *Backends-as-a-Service (BaaS)*, among other names. Developers using such a system write primarily client code, and clients interact with the BaaS through a client library. The main goal of modern BaaS systems is to simplify application development: Developers do not have to design and implement the cloud portions of a distributed app, and they can use familiar client-side programming languages and frameworks.

Recent examples of BaaS systems include Parse [86] and Firebase Realtime Database [46]. However, the overall goal of encapsulating a distributed app's entire backend into a black-box system is one shared by a long lineage of distributed systems research, including the Bayou [100] storage system.

Code offloading. Code offloading reduces energy usage on client devices by executing computationally intensive code on a machine with more resources (e.g., a cloud server). Even if an application originally runs on a single node, code offloading turns it into a distributed application and introduces distributed systems problems that need to be solved. MAUI [21] is a code offload system that dynamically decides which code to offload by profiling device and network conditions and solving an integer linear program based on the application's call graph. MAUI offloads code at the method granularity. It requires developers to label methods that are "remoteable" (i.e., methods that do not interact with device I/O and that can be re-executed without causing external side effects), but the system takes care of figuring out whether those methods actually *should* execute remotely to save energy. In contrast to MAUI, COMET [48] offloads code at the thread rather than method granularity, using distributed shared memory techniques. While MAUI requires developers to annotate their code, COMET supports unmodified applications, although it similarly targets a managed

language (Java in this case) and takes advantage of its features. COMET's scheduler is much simpler than MAUI's, using a simple heuristic that migrates threads from client to server if they have been executing on the client for a while without invoking client-specific code.

Latency hiding. Several distributed systems use speculation to mask the effect of latency on the end-user's experience, providing instantaneous but possibly incorrect feedback in response to user actions. Some of these systems focus on hiding latency for particular classes of applications, using domain knowledge to inform their speculation techniques. Lange, Dinda and Rossoff [63] add client-side speculation to a VNC remote desktop system, while Outatime [66] targets cloud gaming applications. Other systems support arbitrary applications by providing speculative views over a general replicated data store. Many of these systems phrase their abstractions for speculation in terms of eventual consistency: a client's speculative view of data can be considered an eventually consistent view that will converge upon the ground truth if updates quiesce. Bayou [100] provides "separate views of committed and tentative data," where the tentative view represents the speculative execution of updates that have not yet committed. Burckhardt, Leijen, Protzenko, and Fähndrich [14] present Global Sequence Protocol (GSP), a model for replicated data that similarly distinguishes between known and pending updates. As Burckhardt et al. note, "making the updates in the pending queue visible to reads is essential for applications to appear responsive."

1.4 Thesis Systems

This thesis presents three systems that share the common goal of providing developer-friendly interfaces for distributed applications to access high-latency resources. They vary in the portions of the three-tier architecture that they span and in the layers of the software stack that they touch. They also vary in the performance goals that accompany their focus on simplifying development.

Diamond and *Hercules* are distributed storage systems that span the entire three-tier architecture, from the client to the backend. They limit themselves to the library and appli-

cation layers of the software stack. Diamond provides an easy-to-use, transactional interface for accessing shared data in client code, and it also provides the backend features and optimizations required to deliver high throughput with increasing numbers of clients and data sizes. Hercules instead focuses on enabling distributed applications to provide user interfaces that highlight uncertainty in shared state. Hercules identifies and implements an interface that exposes multiple views of shared state, and we show that this interface simplifies developing such a user interface. Exposing uncertainty allows distributed applications to quickly respond to user actions, so Hercules improves performance as reflected in the end-user experience, rather than focusing on scalability.

Marvin is an Android memory manager that performs swapping at the runtime level, in order to let Android devices run more apps simultaneously without needing to kill any due to memory pressure. Whereas Diamond and Hercules are distributed storage systems, Marvin targets the client portion of distributed applications; it co-designs the language runtime and the operating system, reaching deeper into the software stack than either of those systems. In addition to letting Android devices run more apps, Marvin also avoids the user-visible stuttering and delays caused by traditional swapping mechanisms, and by managing memory transparently in the language runtime, it removes the need for developers to explicitly move data back and forth between memory and persistent storage.

Table 1.1 shows how these three systems fit into the diverse body of related work discussed above, organizing systems in terms of their scope across the tiers of a distributed app, their scope across the layers of the software stack, and the problems that they solve. It considers three broad problem categories: simplifying application development, improving performance from the perspective of an individual's user experience (UX), and improving performance in terms of application scalability as the number of users and amount of data increases. Figure 1.2 illustrates where the three systems fit into a distributed application's architecture.

System	Layer within node				Distributed component		System goal		
	Hardware	OS	Runtime	App	Client	Cloud	Simplify development	Performance (UX)	Performance (scalability)
Diamond				x	x	x	x		x
Hercules				x	x	x	x	x	
Marvin		x	x		x		x	x	x
Bayou				x	x	x		x	
FaRM	x	x		x		x			x
MAUI			x	x	x	x	x		x
Outatime				x	x	x		x	
Slicer				x	x	x	x		x
SmartSwap		x			x		x	x	x
Spanner				x		x	x		x

Table 1.1: Categorization of thesis systems and a subset of related work.

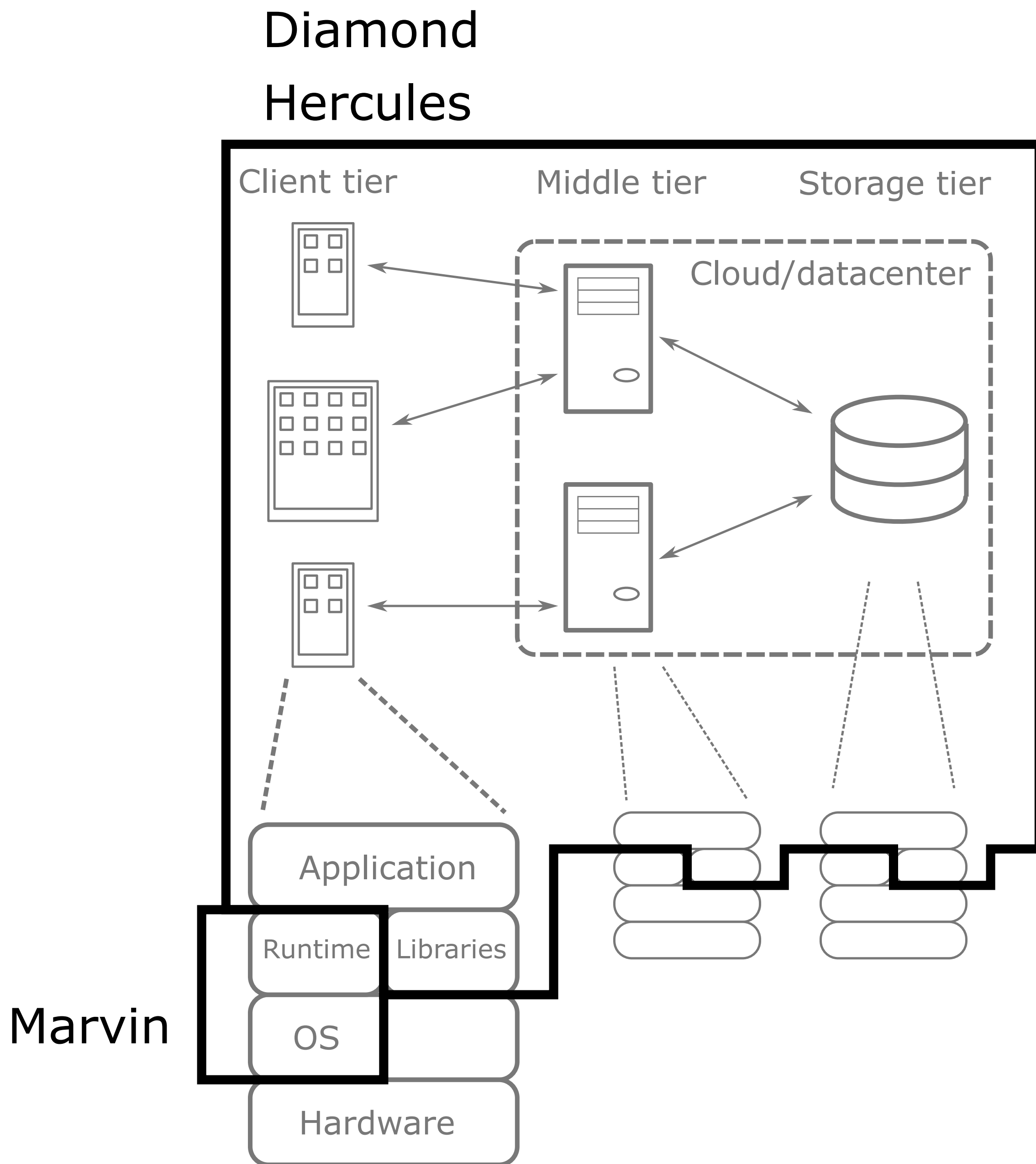

Figure 1.2: The thesis systems' scope across the tiers of a distributed app and the layers of the software stack.

Chapter 2

DIAMOND

This chapter begins our discussion of shared data management with *Diamond*, a data management system for wide-area distributed applications. Traditionally, developers had to invest significant engineering effort into building their apps' backend infrastructure themselves. New Backend-as-a-Service platforms provide infrastructure as a black-box service, letting developers focus on client code, but they expose unwieldy interfaces and force developers to make difficult tradeoffs between consistency and performance. Diamond addresses these problems, providing a simple client interface with strong guarantees and high throughput. This chapter is adapted in part from work published in the 12th USENIX Symposium on Operating Systems Design and Implementation [109].

2.1 Introduction

Today's distributed applications are *reactive* applications that automatically synchronize updates to shared data. Whereas past applications required users to explicitly save their changes or load other users' changes, (e.g., with "save" or "refresh" buttons), reactive apps continually propagate changes and display an up-to-date view of shared state. Document editing apps like Google Docs, games such as Words With Friends, and social media apps like Facebook all have reactive behavior.

Although users expect modern apps to be reactive, distributed storage systems have not kept pace. "Backend-as-a-Service (BaaS)" systems offer interfaces for end-to-end data management, but they provide weak guarantees and force developers to make difficult tradeoffs between programmability and performance. Developers who implement reactive behavior themselves, on the other hand, face the challenge of combining individual components

(database, cache, notification service, etc.) into a full data management system, where the black-box nature of each component makes it harder to provide strong end-to-end guarantees.

Diamond is a distributed data management system that provides strong guarantees, a simple interface, and high performance for reactive applications accessing shared data. Applications read and write shared data using application-level objects and synchronous code inside of transactions with customizable isolation levels. Diamond uses a novel concurrency control mechanism, *data-type optimistic concurrency control (DOCC)*, to provide high throughput when running transactions at strong isolation levels over wide-area deployments. Diamond also provides a novel *reactive transaction* interface for applications to subscribe to shared data updates and make those updates visible to users.

This chapter focuses on Diamond's reactive transactions and a comparison of Diamond to existing BaaS systems. Before diving into those topics, we give an overview of Diamond.

2.2 Diamond Overview

This section describes Diamond's interface, architecture, and transaction protocol.

2.2.1 System Model

A Diamond application consists of processes running on end-user devices and in cloud servers. Diamond itself provides a client library and a set of cloud services that act together to store shared data, mediate concurrent accesses, and deliver notifications. All Diamond applications include developer-defined client code that interacts with the client library. Developers may also choose to implement their own server-side components; these components act as "clients" to Diamond, using the same client library and interface for accessing shared data, but they may perform privileged or computationally intensive tasks.

2.2.2 Reactive Data Types

Diamond represents shared data using *reactive data types (RDTs)*, application-level variables that correspond to shared data items. Diamond transparently synchronizes local modifica-

Type	Operations	Description
Boolean	`Get()`, `Put(bool)`	Primitive boolean
Long	`Get()`, `Put(long)`	Primitive long
String	`Get()`, `Put(string)`	Primitive string
Counter	`Get()`, `Put(long)`, `Increment(long)`, `Decrement(long)`	Long counter
ID	`Generate()`, `Value()`	Unique ID
BooleanList	`Get(index)`, `Set(index,bool)`, `Append(bool)`	Boolean list
LongSet	`Contains(long)`, `Insert(long)`, `Remove(long)`	Long set
LongList	`Get(index)`, `Set(index,long)`, `Append(long)`	Long list
StringSet	`Contains(string)`, `Insert(string)`, `Remove(string)`	String set
StringList	`Get(index)`, `Set(index,string)`, `Append(string)`	String list

Figure 2.1: Reactive data types.

tions to RDTs and updates them to reflect remote changes. RDTs include primitive types such as integers and strings, as well as data types with higher-level semantics like counters, lists, and sets.

Each RDT corresponds to a shared *record* in Diamond's data store that is named by a key in a global namespace. Diamond applications establish the correspondence between an application-level RDT and a shared record using the `rmap()` function. `rmap()` allows client apps written in different languages or frameworks to access the same shared data, and it gives client apps control over which subsets of shared data they are interested in.

2.2.3 Transactions

Diamond applications read and write RDTs inside of transactions. Diamond supports two types of transactions: *read-write* transactions, for modifying shared data, and *reactive transactions*, for reading shared data and propagating changes to the user interface. In this section, we cover read-write transactions; we defer discussion of reactive transactions to Section 2.3.

Read-write isolation level	Reactive isolation level
Strict Serializability	Serializable Snapshot
Snapshot Isolation	Serializable Snapshot
Read Committed	Read Committed

Figure 2.2: Diamond's isolation levels, in order of decreasing guarantees. Each isolation level for read-write transactions corresponds to the adjacent level for reactive transactions.

Applications submit read-write transactions using the `execute_txn()` function, which takes a closure containing the transaction code as an argument. The transaction closure can run application code along with operations on RDTs, but it cannot modify ordinary application variables unless their scope is limited to the closure. This restriction ensures that aborted or re-executed transactions do not have side effects. Diamond runs transactions on a background thread, ensuring that any wide-area accesses do not block the user interface, and it invokes an optional completion callback when a transaction commits.

Diamond supports multiple isolation levels for its transactions, shown in Figure 2.2. Its default level is strict serializability, which simplifies development by removing the need for developers to reason about consistency anomalies; however, developers may choose to use weaker isolation levels to reduce aborts and improve throughput in high-contention scenarios.

2.2.4 Diamond Architecture

Diamond includes a client library that runs in application processes, along with a set of frontend and backend servers that together make up its cloud component. The client library includes a cache. Diamond's backend servers store the ground-truth versions of shared data records, partitioned into shards. For fault tolerance, Diamond assigns each shard to a replica group rather than a single backend server and uses Viewstamped Replication [69] to replicate operations. The frontend servers are stateless processes that communicate directly with Diamond clients. Figure 2.3 illustrates this architecture.

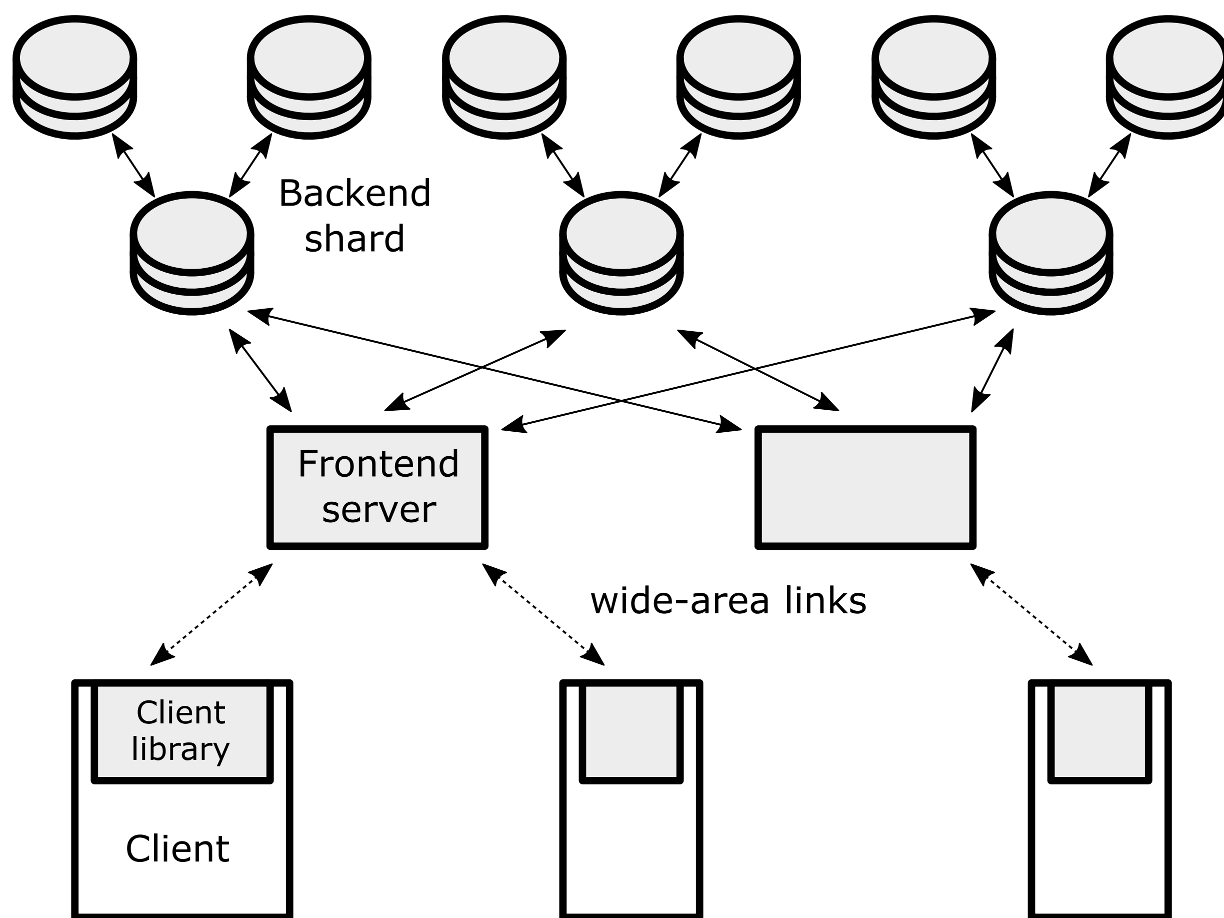

Figure 2.3: **Diamond architecture.** Shaded areas represent Diamond components.

2.2.5 Transaction Protocol

Diamond's transaction protocol consists of two-phase commit layered on top of Viewstamped Replication (VR), like Spanner [20]. It uses 2PC for concurrency control across shards and VR to replicate each individual shard. Unlike Spanner, Diamond uses optimistic concurrency control instead of locking, and it uses a centralized timestamp service to assign timestamps instead of TrueTime.

Traditional optimistic concurrency control (OCC) lets transactions run to completion without acquiring locks and instead checks for conflicts at commit time. Unfortunately, it performs poorly in settings with wide-area network latencies. High latencies increase the duration of each individual transaction, which in turn raises the odds that multiple transactions accessing the same data will overlap and conflict. Excess conflicts result in frequent aborts and a low effective throughput. To avoid this pitfall, Diamond uses a new form of OCC called *data-type optimistic concurrency control (DOCC)*, which uses RDT semantics when checking for conflicts. DOCC allows concurrent transactions to perform commutative operations, such as counter increments or set insertions, without conflicting. Explicitly representing high-level data types in the transaction protocol is the key—if Diamond instead implemented RDTs using reads and writes over an ordinary key-value store, then the low-level operations corresponding to commutative high-level operations would end up conflicting.

Diamond transactions run in two phases, an *execution* and *commit* phase. During the execution phase, the Diamond client library runs a client's transaction closure. The client library locally buffers all writes and modifications to RDTs; it attempts to service reads from its cache but queries the Diamond cloud if necessary. While executing the transaction, it collects an *operation set* containing all RDT operations.

In the commit phase, the client library sends the operation set to a Diamond frontend server. The frontend server, acting as coordinator, executes two-phase commit according to the following steps:

1. The frontend server sends a *Prepare* message to all backend shards containing records

in the operation set.

2. Each backend shard runs a DOCC conflict check. If there are no conflicts, it adds the transaction to a list of prepared transactions and responds *true*; otherwise, it responds *false*.

3. The frontend server requests a commit timestamp from the timestamp service. (As a performance optimization, this step runs concurrently with the previous step.)

4. If all backend shards responded with *true*, the frontend server sends *Commit* messages to all of the backend shards with commit timestamps; if not, it sends *Abort* messages. It then responds to the client library with the transaction result.

Figure 2.4 shows this protocol. The results of the DOCC conflict checks depend on the isolation level being used. Figure 2.5 shows the validation rules for each isolation level.

Diamond's client-side caches may be unavailable when clients crash or disconnect, so Diamond uses multi-versioning instead of explicitly invalidating cache entries. In Diamond's backend storage, each version has a *validity interval* whose start is the timestamp of the transaction that wrote that version, and whose end is the timestamp of the transaction that wrote the next version; if it is the latest version, the end is instead a special *unbounded* value. When the Diamond cloud responds to client read requests, it conservatively bounds validity intervals by replacing unbounded end values with the timestamp of the latest committed transaction that accessed that record. The client library can therefore guarantee that any version in its cache is valid within the validity interval.

2.3 Reactive Transactions

The defining feature of reactive applications is the way they automatically refresh themselves to reflect updates to shared data. Diamond's mechanism for implementing this behavior is *reactive transactions*. A reactive transaction defines a procedure for modifying local application variables and executing local code based on RDT reads. Diamond automatically

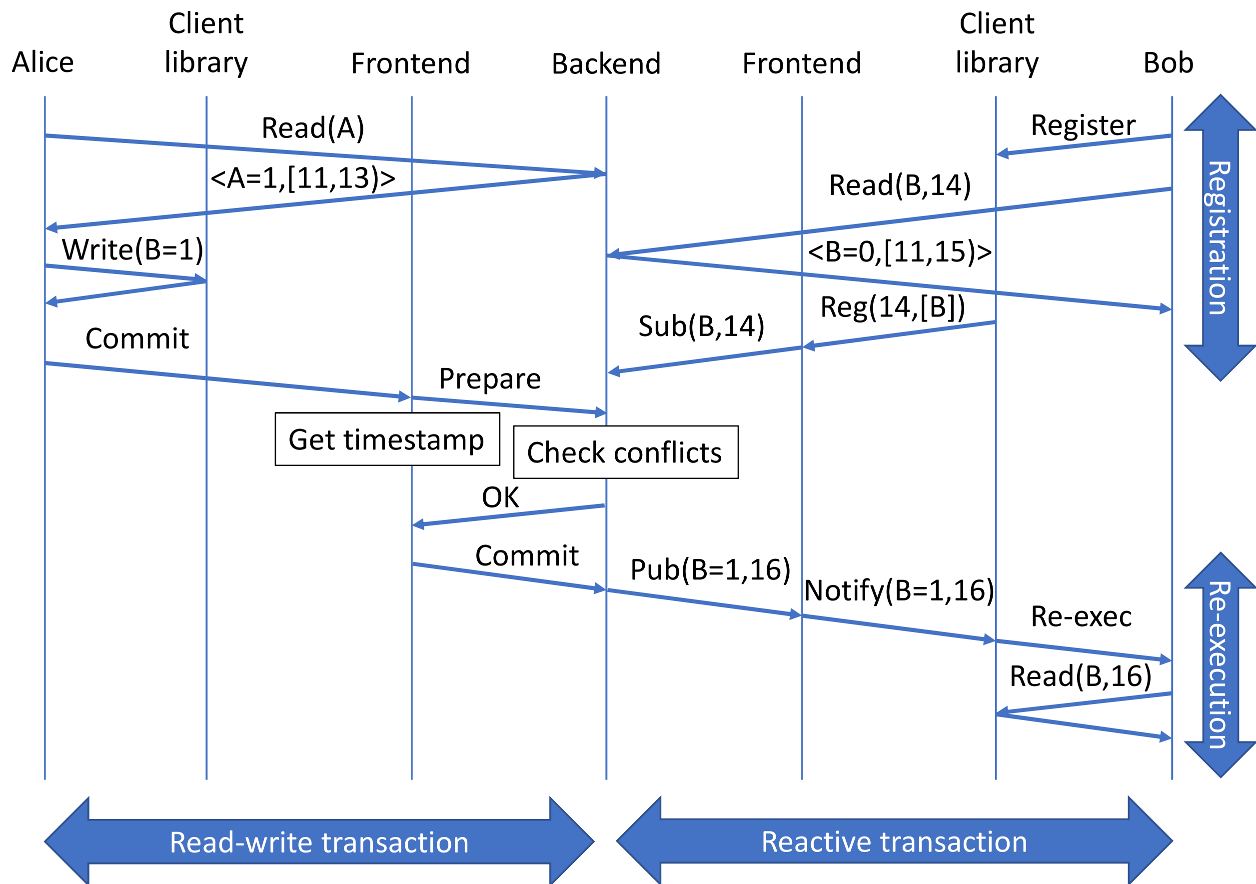

Figure 2.4: **Diamond read-write transaction and reactive transaction protocols.** In this example, Alice runs a read-write transaction that reads record A and writes to record B, while Bob registers a reactive-transaction that reads record B.

Committed op \ Prepared op	Read			Write			Commutative op		
	RC	SI	SS	RC	SI	SS	RC	SI	SS
Read	C	C	C	C	C	A	C	C	A
Write	C	C	A	C	A	A	C	A	A
Commutative op	C	C	A	C	A	A	C	C	C

Figure 2.5: The DOCC validation matrix, specifying whether a transaction with the given operation can commit (C) or must abort (A) in the presence of a conflicting prepared operation. Each prepared operation column is divided into results for the different isolation levels: Read Committed (RC), Snapshot Isolation (SI), and Strict Serializability (SS).

re-executes the reactive transaction whenever its dependent RDTs change, and it ensures that all reads of RDTs in a given execution reflect a single consistent snapshot of shared data. A particularly important use case for reactive transactions is rendering a client app's user interface (UI) based on shared data.

2.3.1 Interface

Applications register reactive transactions by calling the `register_reactxn()` function, which takes a transaction closure as an argument. This closure can read RDTs, but it cannot write to them, to prevent data flow cycles. Aside from that restriction, the closure can perform arbitrary local computation, including invoking functions and modifying variables. These operations are safe because Diamond guarantees that reactive transactions always commit. The Diamond client library returns a transaction ID from `register_reactxn()`, which clients can use to un-register the reactive transaction with the `reactxn_stop()` function.

Diamond guarantees that whenever an RDT that was read inside of a reactive transaction changes, the client library will eventually re-execute the reactive transaction using that updated value. This guarantee applies to the reactive transaction's current read set, which may change from one execution to the next. Diamond also guarantees that each reactive transaction execution reads a consistent snapshot of shared data. Finally, Diamond guarantees that a reactive transaction execution always commits and never aborts.

Diamond's reactive transaction interface has the following benefits, when compared to other distributed storage systems and "backend-as-a-service" systems (we discuss these differences in more detail in Section 2.4):

1. Apps can write synchronous code to update their UI based on shared data, instead of using messy chains of asynchronous callbacks.

2. Reactive transactions can access arbitrary records, rather than being limited to coarse-grained subsets of shared data.

3. Diamond guarantees that each reactive transaction execution reads a consistent snapshot of shared data.

The main drawback of Diamond's interface is that application code must explicitly schedule UI updates onto the UI thread, since reactive transactions run synchronously on a background thread and may block. However, most client-side programming frameworks have straightforward APIs for scheduling code onto the UI thread.

2.3.2 Protocols

Reactive transactions require two protocols: one for registering a reactive transaction, and another for re-executing it when a member of its read set changes. Figure 2.4 illustrates both protocols.

Registration protocol. The registration protocol runs when a client calls `register_reactxn()`. It establishes a mapping in which backend servers associate keys with reactive transactions, sharded by key, and frontend servers map reactive transactions to clients, sharded by client (similar to Thialfi's [3] architecture):

1. The client library assigns a transaction ID to the reactive transaction.

2. It executes the transaction closure at the latest known timestamp.

3. It sends a *Register* request to a frontend server containing the transaction ID, timestamp, and read set.

4. The frontend server creates a record for the reactive transaction that includes a client identifier along with the transaction ID, timestamp, and read set.

5. For each key in the read set, the frontend server sends a *Subscribe* request containing the key and timestamp to the backend shard responsible for that key.

6. The backend shard adds the frontend server to a list of subscribed frontends for that key.

7. If the latest version of the key is at a timestamp after that in the *Subscribe* request, the backend shard immediately runs the re-execution protocol below at the latest timestamp.

Re-execution protocol. The re-execution protocol runs whenever a read-write transaction commits. It is responsible for re-running any reactive transactions whose read sets were modified by the newly committed transaction:

1. For each modified key, its backend shard sends a *Publish* message containing the commit timestamp to every frontend server subscribed to that key.

2. The frontend server looks up the reactive transactions with read sets containing that key; for each one, if the commit timestamp is after the timestamp of the last notification sent to that client, the front-end server sends a *Notify* message to the client containing the commit timestamp and transaction ID.

3. The client re-executes the reactive transaction, performing all reads at the commit timestamp.

4. If the read set has changed, the client sends another *Register* request to its frontend server.

The client library's use of a particular timestamp when re-executing a reactive transaction allows it to both guarantee that the reactive transaction reads a consistent snapshot and that the reactive transaction always commits.

2.3.3 Data Push Notifications

A reactive transaction executes synchronously, so if the client cache is missing the records in its read set, the transaction might require several wide-area round trips to perform all its reads. To improve performance, the Diamond cloud takes advantage of the fact that it knows the reactive transaction's read set, and it bundles data along with its notification messages to clients. We refer to this optimization as *data push notifications*.

As described in the re-execution protocol above, whenever a read-write transaction modifies a key in a reactive transaction's read set, a frontend server sends a *Notify* message to its client telling it to re-run the reactive transaction at the read-write transaction's commit timestamp. Before doing so, the frontend server additionally performs snapshot reads of the keys in the reactive transaction's read set at the commit timestamp, and it bundles those values in the *Notify* message. The client adds those values to its cache, and when it re-executes the reactive transaction, it can read those values without contacting the Diamond cloud. It only needs to perform wide-area reads if the read set has changed.

Data push notifications are possible because Diamond integrates the notification service and data store into a single system. Standalone notification services such as Thialfi [3] treat notifications as opaque messages, so apps must perform additional wide-area round trips to retrieve the associated application data.

2.4 Comparison with BaaS Systems

By encapsulating the cloud portions of a distributed app and letting the developer only write client code, Diamond broadly resembles backend-as-a-service (BaaS) systems such as Firebase Realtime Database [46] and Parse [86]. Designing the client interface for such systems is difficult, and we argue here that Diamond's interface is superior to those of its competitors.

2.4.1 The Difficulty of Designing a BaaS Interface

BaaS systems provide fine-grained interfaces for reading and writing shared data in the client rather than the server, creating new challenges when designing that interface. This section explains this shift and the difficulties it presents.

Application operations and reactive updates. Distributed app clients perform two broad types of actions: *application operations*, which translate user actions into shared data modifications, and *reactive updates*, which communicate shared data changes to the user.

Application operations read and write shared data according to application logic. One application operation may involve multiple reads and writes of shared data, as well as reads of local client state. For instance, the operation to buy a pet in a game may involve reading the amount of money in the user's wallet (shared data), reading which pet is selected in the client UI (local client state), writing a smaller amount of money to the user's wallet (shared data), and writing the acquisition to the user's inventory list (shared data).

Reactive updates automatically modify the client's UI to reflect changes to shared data. A reactive update reads shared data and writes local client state, and it might read local client state as well. Going off of the previous game example, a reactive update responsible for refreshing a user's inventory menu might read the amount of money in their wallet (shared data), write the updated amount to a UI text box (local client state), read which inventory sub-menu is currently open (local client state), and, if the "pets" sub-menu is open, read the identity of the new pet (shared data) and write an icon of the pet to an entry in the sub-menu (local client state).

A shift in the interface. Each application operation and reactive update corresponds to multiple fine-grained reads and writes of shared data, and BaaS systems differ from the three-tier architecture in where that fine-grained interface is located.

In the traditional three-tier architecture, clients perform application operations by sending a single message to a middle-tier server. The server then calls into the storage tier to

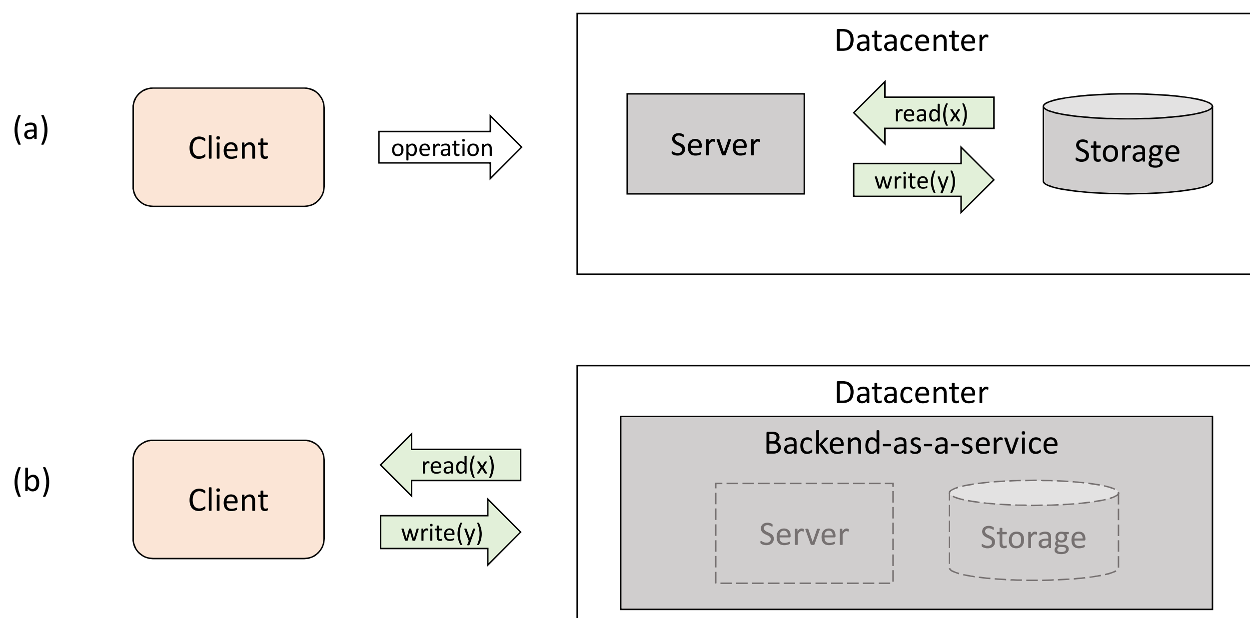

Figure 2.6: In distributed apps built with the three-tier architecture (a), clients send coarse-grained messages to the middle tier, whereas in apps built with BaaS systems (b), clients directly read and write shared data.

perform the multiple reads and writes of shared data required to execute the operation. Similarly, for a reactive update, the server reads all the necessary shared data from the storage tier and then sends a single message to the client.

BaaS systems move the interface for fine-grained shared data manipulation from the server to the client. They expose client interfaces for directly reading and writing shared data. Figure 2.6 illustrates this shift. Moving the interface to the client means that developers can define application logic directly in client code, fitting with BaaS systems' goal of letting developers not worry about servers and backend components.

Interface goals and challenges. BaaS client interfaces have three high-level goals:

1. *Maintain a responsive UI.* Most client-side programming frameworks run code on the UI thread by default, allowing that code to react to input events and modify the display. In order to keep the UI responsive, these frameworks strongly encourage developers to

avoid executing long-running blocking operations on the UI thread. A BaaS system should make sure that its client interface does not block the UI thread.

2. *Provide strong transactional guarantees.* Many apps define application operations whose shared data reads and writes must be executed as a single atomic unit to maintain app-level invariants and ensure correctness. (E.g., in the game example above, the app should decrement the user's money and add the pet to their inventory in a single atomic operation.) Similarly, reactive updates often read multiple shared data items, and the UI will be incorrect if those reads do not reflect a single consistent snapshot. The interface should allow the application to group multiple related reads and writes and have them execute as a single transactional unit.

3. *Be intuitive for developers.* The interface should be clean and easy-to-use, avoiding clunky nested callbacks or manual thread scheduling.

When the interface for performing fine-grained shared data reads and writes moves from servers to clients, providing all three properties at once becomes more difficult. Clients are a wide-area round trip away from the underlying storage system, so each read and write becomes a long-running, blocking operation. In order to maintain a responsive user interface, these reads and writes must therefore be asynchronous or run on a background thread. Asynchronous interfaces must be designed with care in order to support transactions and remain intuitive. As the discussion below shows, existing interfaces succeed in maintaining a responsive UI, but they do not support transactional guarantees, and they are not intuitive. Diamond, on the other hand, meets all three requirements.

2.4.2 Application Operations

Existing BaaS systems provide a variety of interfaces for executing application operations, but none of these interfaces are intuitive or enable strong guarantees. These interfaces generally suffer due to their use of asynchronous callbacks and coarse-grained references to

shared data. "Coarse-grained references" mean that the multiple data items involved in an application operation must be read and written using separate API calls. If each of these API calls are asynchronous, a single application operation that reads and writes multiple data items involves multiple callbacks, making code unintuitive and difficult to read. Furthermore, because data accesses are spread across callbacks, the interface does not provide a way to perform multiple data accesses as a single transactional unit.

In the following discussion, we use the example of a game in which players buy pets to illustrate the interfaces of the systems. In this example, each player has a name, an amount of money, and the name of his or her currently owned pet. Each pet has a cost and the name of its current owner. Buying a pet consists of updating the player's pet variable to be the name of the pet, updating the player's money to subtract the cost of the pet, and updating the pet's owner to be the name of the player. Although this operation seems simple, because it involves multiple reads and writes across two different types of objects, it is surprisingly difficult to implement using existing BaaS interfaces.

Parse. Due to its reliance on asynchronous callbacks and per-class queries, Parse's interface is unintuitive and cannot provide strong guarantees. Parse defines an application object called the `ParseObject` corresponding to a shared JSON object. Writes to shared data in Parse are split into two steps: a synchronous write to the local `ParseObject` (using the `put()` method) and an explicit asynchronous propagation of the changes in the local `ParseObject` to the shared data in the BaaS (using the `saveInBackground()` method). Reads of shared data are similarly split into an asynchronous fetch and a synchronous local read: given an existing `ParseObject`, the asynchronous `fetchInBackground()` method takes a callback whose argument is an updated copy of the `ParseObject`, and synchronous local reads using the `get()` method or its typed variants can happen inside of the callback. Parse does not support any form of transactions over shared data, and the asynchronous saves and fetches can only be performed on a per-object basis.

In order to allow clients to reference shared data, Parse provides an interface for per-

forming queries over `ParseObjects`: the app specifies a set of constraints to a `ParseQuery` object, and its asynchronous `findInBackground()` method invokes a callback with copies of all the `ParseObjects` of a given class matching the constraints. Parse does not organize data hierarchically. Once a client creates a `ParseObject`, it is assigned a random unique ID and inserted into a flat namespace containing all objects of that class. One `ParseObject` can reference another, but the referenced object must be fetched using a separate asynchronous method call and callback.

Parse's interface makes complex application operations cumbersome and prohibits the service from providing transactional guarantees. Because Parse queries can only retrieve objects of one user-defined class at a time, and a `ParseObject` referenced by another object must be fetched in a separate callback, implementing application operations that read multiple objects of different classes is difficult. The most straightforward way of implementing such an application operation is for the client to issue a series of nested fetches or queries for each required `ParseObject`, executing the next query inside the callback of the previous one and executing the application operation inside the final callback. This approach results in deeply nested, difficult-to-read code. The client code must also explicitly call `saveInBackground()` on any `ParseObject` that was updated during the application operation, potentially hampering readability or introducing bugs. Parse does not provide any guarantees of atomicity or isolation for reads and writes on shared data, and its interface makes such guarantees impossible. Because wide-area reads from and writes to different `ParseObjects` happen in separate asynchronous method calls, there is no way to group multiple reads or writes together into one transactional unit.

Listing 2.1 shows the code used to implement the pet game application operation in Parse. The first `ParseQuery` finds the pet that the player wants to buy, and a second query inside the callback of the first finds the Player object corresponding to the client's user. The logic to perform the writes of the application operation happens inside the callback of the second query.

Listing 2.1: The pet game application operation in Parse.

```
String playerName;

public void buyPet(String petName) {
  ParseQuery<ParseObject> petQuery = ParseQuery.getQuery("Pet");
  petQuery.whereEqualTo("name", petName);
  petQuery.findInBackground(new FindCallback<ParseObject>() {
    public void done(List<ParseObject> objects, ParseException e) {
      final ParseObject pet = objects.get(0);
      ParseQuery<ParseObject> playerQuery = ParseQuery.getQuery("Player");
      playerQuery.whereEqualTo("name", playerName);
      playerQuery.findInBackground(new FindCallback<ParseObject>() {
        public void done(List<ParseObject> objects, ParseException e) {
          ParseObject player = objects.get(0);
          player.set("money", player.getInt("money") - pet.getInt("cost"));
          player.set("pet", petName);
          pet.set("owner", player.getString("name"));
          player.saveInBackground();
          pet.saveInBackground();
        }
      });
    }
  });
}
```

Firebase Realtime Database. Firebase's application operation interface also suffers from its reliance on asynchronous callbacks, as well as the hierarchical granularity of its methods for accessing shared data. Firebase defines a set of application objects that represent shared JSON objects. These JSON objects are hierarchically organized, and a reference to one object can be used to read or modify its children. However, the Firebase documentation [45] advises developers to denormalize or "flatten" data stored in Firebase, rather than use deeply

nested layouts, for performance and security reasons: any read of an object reads all of its descendants as well, and permissions applied to an object apply to all of its descendants, so a deeply nested data layout both makes reads more costly and makes permissions harder to manage.

Firebase provides an interface for executing transactions on single JSON hierarchies and an interface for executing asynchronous writes across hierarchies without any transactional guarantees. Firebase does not provide an interface for performing reads of shared data outside of transactions, but the developer has the option of maintaining separate local copies of shared data and keeping them up-to-date via Firebase's event listener mechanism (described in Section 2.4.3). Each Firebase transaction executes on a single JSON object, allowing reads and writes of that object and its descendants. In a Firebase transaction handler, the client defines an update function that takes a `MutableData` object corresponding to a shared JSON object as an argument and performs one or more reads and writes on the object. Firebase runs the update function locally and then attempts to apply the changes in the local `MutableData` object to the shared data in the BaaS. If another client has concurrently written to the same data, and its state is no longer the same as it was on the client at the start of the transaction, Firebase reruns the update function (with the `MutableData` object now holding the new version of the data) until it succeeds in applying the local changes to a version of the shared data matching the original local version. Non-transactional (or "blind") writes update the local copies of JSON objects synchronously, and Firebase propagates the changes asynchronously to the shared data in the BaaS on a best-effort basis. Clients perform blind writes using the `updateChildren()` method. Blind writes can be *multi-path* writes that simultaneously update multiple objects across different JSON hierarchies.

Complex application operations in Firebase are not intuitive, and the guarantees that they provide are limited, due to the restrictions of Firebase's transaction interface. Because Firebase transactions can only read and write data that hierarchically descends from a single JSON object, but a deeply nested data layout is discouraged, Firebase transactions are limited in power: a flat data layout increases the likelihood that the shared data corresponding

to one application operation is spread across JSON objects and so cannot be atomically updated in one transaction (running all transactions on the root reference of the database is possible, but it would severely hurt performance). In order to perform an application operation that extends across multiple JSON hierarchies, the developer must maintain separate local copies of shared data as described above, and then either use separate transactions to write to each of the JSON hierarchies or write to all of the hierarchies simultaneously with one blind write on the root database reference. Both of these choices are unintuitive. The transaction interface provides fairly strong atomicity and isolation guarantees, guaranteeing that the shared data will eventually reflect the result of running the update method atomically once on a consistent snapshot of the data, but these guarantees do not apply if the application operation spans multiple JSON hierarchies. When performing application operations by using blind writes and separate local copies for reads, multi-path blind writes appear to guarantee atomic execution, but there is no way to guarantee that these writes are atomic with the reads that populated the local copies of data.

Figure 2.2 shows the code required to implement the pet game application operation in Firebase. This code uses one multi-path blind write to update the player's two variables and the pet's owner in a single method call, and it uses separate local copies of shared data to perform the reads required for the operation. These local copies must be kept up-to-date by the reactive update code (discussed in Section 2.4.3), and as mentioned above, there's no guarantee of atomicity between when the local variables were updated and when the write occurs. The burden of maintaining separate local copies of shared data is particularly noticeable when it comes to the costs of the pets—without a mechanism to perform explicit reads, in order to make sure the application operation can handle buying any possible pet, the code must maintain its own local data structure (`petCosts`) mapping pet names to their costs and use this data structure in the application operation.

Listing 2.2: The pet game application operation in Firebase.

```
String playerName;
FirebaseDatabase database;
```

```
Map<String, int> petCosts; // Local copies of shared data
int money;                 // Kept up to date by EventListeners (not shown)

public void buyPet(String petName) {
    DatabaseReference dbRef = database.getReference();
    Map<String, Object> updates = new HashMap<>();
    updates.put("/players/" + playerName + "/money",
                money - petCosts.get(petName));
    updates.put("/players/" + playerName + "/pet", petName);
    updates.put("/pets/" + petName + "/owner", playerName);
    dbRef.updateChildren(updates);
}
```

Diamond. With Diamond's interface, developers write synchronous code that executes on a background thread instead of asynchronous callbacks, making application code simpler and allowing Diamond to group shared data accesses into transactions. A Diamond transaction is not tied to a particular shared data item or subtree, allowing developers to implement application logic without needing to denormalize shared data or worry about performance tradeoffs.

Listing 2.3 shows a Diamond implementation of the pet game application operation. Once the developer has `rmap()`'d the required variables, they only need to write a single transaction that contains simple `Put()`s and `Get()`s in a natural, imperative style.

Listing 2.3: The pet game application operation in Diamond.

```
String playerName;

public void buyPet(String petName) {
    DLong playerMoney = rmap(playerName + ":money");
    DString playerPet = rmap(playerName + ":pet");
    DLong petCost = rmap(petName + ":cost");
```

```
    DString petOwner = rmap(petName + ":owner");

    execute_txn(() -> {
        playerMoney.Put(playerMoney.Get() - petCost.Get());
        playerPet.Put(petName);
        petOwner.Put(playerName);
    });
}
```

2.4.3 Reactive Updates

Reactive update interfaces for existing BaaS systems suffer from the same weaknesses as their application operation interfaces. They allow responsive UI programming, but they do not enable strong transactional guarantees and are not intuitive, particularly in the case of complex mappings between shared data and derived local state (e.g., where one local variable is derived from multiple pieces of shared data). By and large, the causes of these weaknesses are the same as well: the interfaces rely on asynchronous callbacks and only allow coarse-grained data accesses.

We will compare the reactive update interfaces for each of the systems by considering a different problem within the context of the pet game example. Here we suppose that the game shows two text boxes, one with the name of the pet currently owned by the player, and another with the player's net worth (the player's money added to the cost of the pet). The player's net worth represents a more complex mapping of shared data to local state, as it is a local variable derived from two different items of shared data. The game uses reactive updates to refresh these text boxes whenever the underlying data changes.

Parse. As with application operations, the use of asynchronous callbacks and per-class queries in Parse's reactive update interface makes it unintuitive and prevents it from providing strong guarantees. Parse supports reactive updates through its *LiveQuery* interface. LiveQuery allows clients to subscribe to queries and register callbacks that execute every

time the results of those queries change. To use LiveQuery, a client creates a `Subscription` object by calling `subscribe()` on a `ParseQuery`. The client then calls `Subscription` methods to register callbacks for different types of events, such as `create` (a new `ParseObject` matching the query has been created), `update` (an existing object matching the query has been updated), and `enter` (an existing object that previously did not match the query has been updated to match the query).

Parse's reactive update interface does not easily support complex mappings between shared data and local state, and it provides weak and unclear guarantees. LiveQuery subscriptions work well for updating local state that is derived from only one `ParseObject` or class of `ParseObjects`. However, if a piece of local state is derived from multiple classes of objects, one subscription alone cannot deliver all the information required to make an update. In this case, the client has several choices for how to implement updates, none of which are ideal. For instance, the client could register a subscription for each relevant class of `ParseObject`, and inside each of those subscription callbacks, trigger a method that issues a series of nested queries for all of the required objects and updates the local state. Alternatively, the client could maintain separate local copies of the shared data used to derive the local state, and inside each of the subscriptions, update the local copies of the objects returned by that subscription and then trigger code to recompute the local state based on the local copies of the shared data. Aside from being unintuitive, neither of these options allow any guarantee of atomicity for the multiple reads involved in the update, since the reads are split across separate callbacks.

Listing 2.4 shows the code required to implement reactive updates for the pet game in Parse (the Parse LiveQuery API for Android has not yet been released, so this code represents the author's guess at what the API will look like, based on the released JavaScript and iOS APIs). In this example, one LiveQuery subscription listens for updates to the Player `ParseObject`, and another listens for updates to Pet `ParseObjects`. The displayed pet name is only derived from one shared data item, so its update code is straightforward, but since the player's displayed net worth is derived from two `ParseObjects` of different classes, the code

to update net worth is more complicated. The net worth update code is factored out into its own method, because the update is triggered by changes to either the player or the pet object, and this method performs two queries of its own to retrieve the two shared objects in the same scope. Since the `updateNetWorth()` method is called by either the Player or the Pet subscription, one of the two queries in the method is redundant; the method could be rewritten to eliminate the redundant query at the cost of additional code complexity. Also, note that as written, this code calls `updateNetWorth()` whenever *any* pet is updated, because the client does not know which pet is owned by the player. By keeping a separate copy of the name of the player's pet, a shared variable, the client could only subscribe to updates for the pet owned by the player, at the cost of explicitly keeping the local copy of the pet name up to date.

Listing 2.4: Reactive updates for the pet game in Parse.

```
String playerName;

TextView netWorthTextBox; // UI elements displaying shared data
TextView petTextBox;

// This method is called once when the client is initialized
public void setupReactiveUpdates() {
  ParseQuery<ParseObject> playerQuery = ParseQuery.getQuery("Player");
  playerQuery.whereEqualTo("name", playerName);
  Subscription playerSub = playerQuery.subscribe();
  playerSub.handle(Event.Updated, new EventHandler() {
    public void handle(final ParseObject player, ParseException e) {
      String petName = player.getString("pet");
      petTextBox.setText(petName);
      updateNetWorth();
    }
  });
  ParseQuery<ParseObject> petQuery = ParseQuery.getQuery("Pet");
  Subscription petSub = petQuery.subscribe();
```

```
  petSub.handle(Event.Updated, new EventHandler() {
    public void handle(final ParseObject pet, ParseException e) {
      updateNetWorth();
    }
  });
}

public void updateNetWorth() {
  ParseQuery<ParseObject> playerQuery = ParseQuery.getQuery("Player");
  playerQuery.whereEqualTo("name", playerName);
  playerQuery.findInBackground(new FindCallback<ParseObject>() {
    public void done(List<ParseObject> objects, ParseException e) {
      final ParseObject player = objects.get(0);
      String petName = player.getString("pet");
      ParseQuery<ParseObject> petQuery = ParseQuery.getQuery("Pet");
      petQuery.whereEqualTo("name", petName);
      petQuery.findInBackground(new FindCallback<ParseObject>() {
        public void done(List<ParseObject> objects, ParseException e) {
          ParseObject pet = objects.get(0);
          netWorthTextBox.setText(player.getInt("money")
                                  + pet.getInt("cost"));
        }
      });
    }
  });
}
```

Firebase Realtime Database. The reactive update interface in Firebase is unintuitive and does not provide strong guarantees due to its use of asynchronous callbacks and the hierarchical granularity of its data accesses. Firebase supports reactive updates by defining interfaces for event listeners that can be attached to an application object. These listeners are triggered each time the corresponding JSON shared data object changes. The listeners run

a client-specified callback that receives a `DataSnapshot` object with a copy of the updated data. The client can propagate changes to local state inside of the callback.

The main limitation of Firebase's event listener interface is that each event listener is tied to one JSON hierarchy. This limitation makes it difficult to update a UI element or other piece of local state that is derived from multiple pieces of shared data spread across JSON hierarchies. To keep such a piece of local state updated, the client must trigger the update of the local state inside the event listeners for each of the shared data items used to compute it. Furthermore, since the `DataSnapshot` objects are only accessible inside the scope of the event listeners, the client must maintain a separate local copy of each relevant shared data item and use those local copies when re-computing the value of the local state. In addition to making the interface unintuitive, the tying of event listeners to individual JSON objects prevents the interface from providing any guarantees of isolation: the values of shared data used to re-compute a derived local variable may not represent a consistent snapshot of the data.

Listing 2.5 shows the code required to perform reactive updates for the pet game in Firebase. Note that in addition to updating the text boxes, the event listeners are also responsible for updating the local copies of shared data. These local copies are used both for reads performed during the application operation of buying a pet and for reads performed when updating the player's net worth, a local variable derived from multiple shared data items. As in the Parse example, the method to update the player's net worth is factored out into a separate method, since the update must be triggered when either the player or the pet changes.

Listing 2.5: Reactive updates for the pet game in Firebase Realtime Database.

```
String playerName;
FirebaseDatabase database;

Map<String, int> petCosts; // Local copies of shared data
int money;
```

```
String petName;

TextView netWorthTextBox; // UI elements displaying shared data
TextView petTextBox;

// This method is called once when the client is initialized
public void setupReactiveUpdates() {
  DatabaseReference petsRef = database.getReference("pets");
  petsRef.addChildEventListener(new ChildEventListener() {
    public void onChildAdded(DataSnapshot snapshot, String prevName) {
      String name = snapshot.getChild("name").getValue();
      int cost = Integer.parseInt(snapshot.getChild("cost").getValue());
      petCosts.put(name, cost);
    }
    public void onChildChanged(DataSnapshot snapshot, String prevName) {
      String name = snapshot.getChild("name").getValue();
      int cost = Integer.parseInt(snapshot.getChild("cost").getValue());
      petCosts.put(name, cost);
      if (name.equals(petName)) {
        updateNetWorth();
      }
    }
  });
  DatabaseReference playerRef = database.getReference("players/" + playerName);
  playerRef.addValueEventListener(new ValueEventListener() {
    public void onDataChange(DataSnapshot snapshot) {
      money = Integer.parseInt(snapshot.getChild("money").getValue());
      petName = snapshot.getChild("pet").getValue();
      petTextBox.setText(petName);
      updateNetWorth();
    }
  });
}
```

```
public void updateNetWorth() {
  netWorthTextBox.setText(money + petCosts.get(petName));
}
```

Diamond. Diamond's approach of running synchronous code on a background thread simplifies the code required to implement a reactive update, and it allows Diamond to read a single, consistent snapshot of shared data when performing a single logical update. Listing 2.6 shows how a developer might implement the pet game reactive update in Diamond. The one drawback of Diamond's interface is that because it runs code on a background thread, developers must explicitly schedule UI accesses onto the UI thread. Most UI frameworks have simple interfaces for doing so (e.g., the `runOnUiThread()` method in this Java/Android example), so we believe that the simplicity of synchronous code is worth this additional requirement.

Listing 2.6: Reactive updates for the pet game in Diamond.

```
String playerName;

TextView netWorthTextBox; // UI elements displaying shared data
TextView petTextBox;

// This method is called once when the client is initialized
public void setupReactiveUpdates() {
    register_reactxn(() -> {
        DString playerPet = rmap(playerName + ":pet");
        DLong playerMoney = rmap(playerName + ":money");
        DLong petCost = rmap(playerPet.Get() + ":cost");

        String petName = playerPet.Get();
        int netWorth = playerMoney.Get() + petCost.Get();
```

```
        runOnUiThread(() -> {
            petTextBox.setText(petName);
            netWorthTextBox.setText(netWorth);
        });
    });
}
```

2.5 Implementation

Our Diamond prototype includes frontend and backend servers, a native client library, and Java and Python bindings for clients. We used javacpp [16] for the Java bindings and Boost.Python [1] for the Python bindings. We implemented the frontend and backend servers and client library in 11,795 lines of C++ code; the Java bindings are 939 lines of code, and the Python bindings are 115 lines. We cross-compiled Diamond and its dependencies for Android using an NDK standalone toolchain [47].

Our prototype implements all of the reactive data types in Figure 2.1, although not all of them support DOCC. Our prototype does not support client-side persistence, and the cloud store uses in-memory replication as its persistence mechanism.

2.6 Evaluation

We evaluated Diamond's performance as well as its ability to simplify programming distributed apps. Our evaluation showed that Diamond provides stronger guarantees and reduces app complexity when added to existing apps, that it lets developers quickly write new distributed apps, and that it performs data management with high performance.

2.6.1 Programming Experience

This section describes our experience programming a Diamond app from scratch and using Diamond to replace an existing app's custom data management.

Chat Room. We built an Android chat room app on top of Diamond, along with a baseline version that implements its own shared data management. Our baseline app includes a web server that interacts with clients using a REST [36] API. Clients send `POST` requests to send messages, and they use `GET` requests to poll the server for new messages every second and update the display. The web server is implemented in Java using Jetty [37] and uses Redis [91] for storage. The Diamond version represents the chat log as a `DStringList`, uses a read-write transaction to append new messages to the log, and uses a reactive transaction to listen for new messages and redraw the display.

Diamond eliminated the need to implement a backend/server component entirely, shrinking the app's code count by 33% (355 vs. 225 LoC). Diamond also added isolation, by ensuring that all clients see a consistent view of the chat log, and reactivity, by eliminating the need for clients to poll for changes.

PyScrabble. To evaluate the impact of replacing an existing app's shared data management with Diamond, we built a Diamond version of PyScrabble [19], an open-source multiplayer Scrabble game. The original PyScrabble uses a centralized server to process moves and notify players. The client updates local state as the player puts tiles on the board, and then it sends a single message to the server once the player finishes their move. Scrabble is a turn-based game, so the server provides strong isolation by tracking turns and only accepting moves from the client whose turn it is. However, the server does not store data persistently, so a single failure wipes out all app data.

We modified PyScrabble to use Diamond to store and access its shared data. Our main task was figuring out how to adapt PyScrabble to Diamond's transaction model. We chose to directly `rmap` the Scrabble board to reactive data types and update the UI in a reactive transaction, so our modified client has to commit a transaction for each user action to make it visible to the user. As a result, other users can see a player lay down tiles in real-time, rather than only seeing one update at the end of the move as in the original design. Adding Diamond to PyScrabble reduced the 8700-line app's size by 13%, simplified its structure by

removing the server, and added fault tolerance, while retaining strong isolation.

2.6.2 Performance Evaluation

We ran experiments to compare Diamond's throughput to a typical built-from-scratch data management solution and determine the performance overhead of stronger isolation levels. We also measured the performance benefits of two optimizations, DOCC and data push notifications. Our results showed that Diamond provides performance competitive with custom-built apps, that DOCC improves performance when running with strong isolation, and that data push notifications reduce the latency of reactive updates.

Experimental setup. We ran experiments on Google Compute Engine virtual machines. Our Diamond deployment included 16 frontend servers and five backend partitions in the US-Central region. Each backend partition contained three replicas spread across different availability zones in the same region. We put clients in the US-East region. The latency between availability zones was ≈1ms, and the latency between regions was ≈36ms.

We used a benchmark based on Retwis [89], a Twitter clone that uses Redis for backend storage. Our benchmark implements a Twitter-like workload with control over contention. It runs a mix of five transactions that include 4–21 operations, according to the following distribution: loading a user's home timeline (50%), posting a tweet (20%), following a user (5%), creating a new user (1%), and "liking" a Tweet (24%). We used 100K keys and a Zipf distribution with a co-efficient of 0.8 to measure performance in high-contention scenarios.

Baseline performance. We compared Diamond to a baseline implementation of the Retwis benchmark that performs its own data management according to a typical three-tier design. Our baseline implementation uses Redis for backend storage and Jetty web servers that receive client requests and translate them to Redis reads and writes. The Jetty servers use Redis's `WAIT` command to synchronously replicate writes, matching Diamond's fault-tolerance guarantees; however, `WAIT` does not enforce atomicity or isolation across the

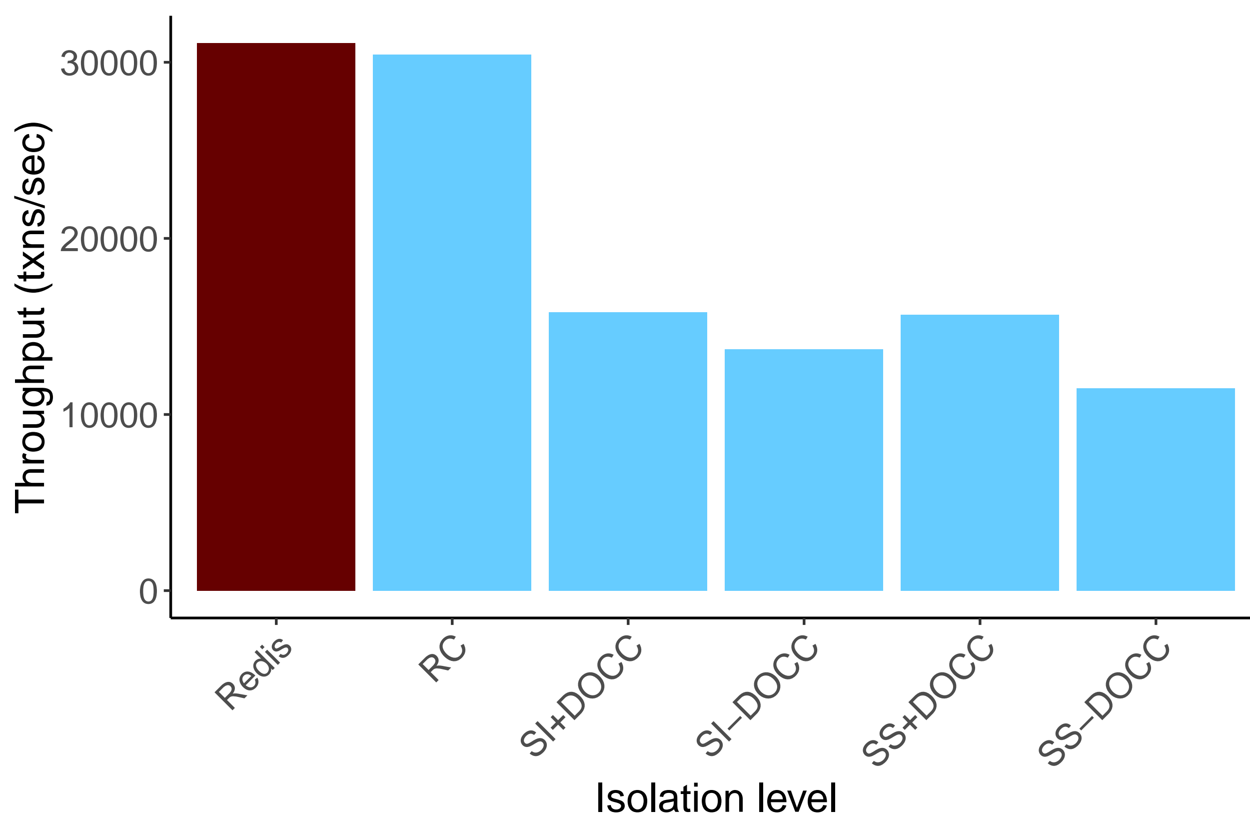

Figure 2.7: Peak throughput of Diamond compared to our baseline Redis/Jetty implementation. Diamond's isolation levels are Read Committed (RC), Snapshot Isolation (SI), and Strict Serializability (SS).

multiple accesses in a Retwis transaction.

Figure 2.7 compares Diamond's peak throughput at different isolation levels to that of the baseline implementation. In all cases, we ran enough clients to saturate the backend and ensure that backend servers were the bottleneck. Diamond's Read Committed isolation level delivered a peak throughput of 30.5K transactions/second, slightly lower than the baseline implementation's 31K transactions/second. Read Committed does not isolate transactions, but it does guarantee a single global operation ordering, which Redis does not; in addition, Diamond in Read Committed mode still provides reactive transactions. Diamond's weakest isolation level therefore simplifies development with very low overhead.

At stronger isolation levels, Diamond's throughput decreases. Snapshot Isolation and Strict Serializability each had around 50% lower throughput than Read Committed. This drop is the result of aborts due to conflicting transactions as well as added message traffic required by two-phase commit. Removing DOCC caused a further 27% drop and 13% drop for Snapshot Isolation and Strict Serializability respectively. The loss of DOCC hits Strict Serializability harder because commutative operations reduce to writes, which can commit with conflicting reads in Snapshot Isolation but not in Strict Serializability. Diamond's stronger isolation levels expose a tradeoff to developers, eliminating consistency anomalies at the cost of lower performance.

Benefit of DOCC. DOCC's effect depends on transaction length, contention, and the amount of commutative operations. We expect DOCC to significantly reduce aborts and improve throughput for long transactions with many commutative operations, and have a smaller effect on shorter transactions with a higher proportion of reads and writes. We measured the throughput improvement for each transaction type in our benchmark. Figure 2.8 shows the results.

The `post_tweet` transaction appends a new tweet to a user's own timeline and to the timelines of all their followers. Each user has between 5 and 20 followers; when multiple users with shared followers post new tweets, the append operations conflict and cause aborts without DOCC. As expected, DOCC results in a huge throughput improvement (25x) for `post_tweet` transactions. DOCC also meaningfully but less dramatically improves the throughput of the shorter `add_follower` (3.7x), `add_user` (1.9x) and `like` (2x) transactions. The `get_timeline` transaction is read-only, so DOCC does not affect its abort rate, but it does get a small throughput improvement (2.6%) due to the reduced server load caused by the lower abort rates of other transactions. These results show that DOCC is crucial for running large transactions at high isolation levels with good performance.

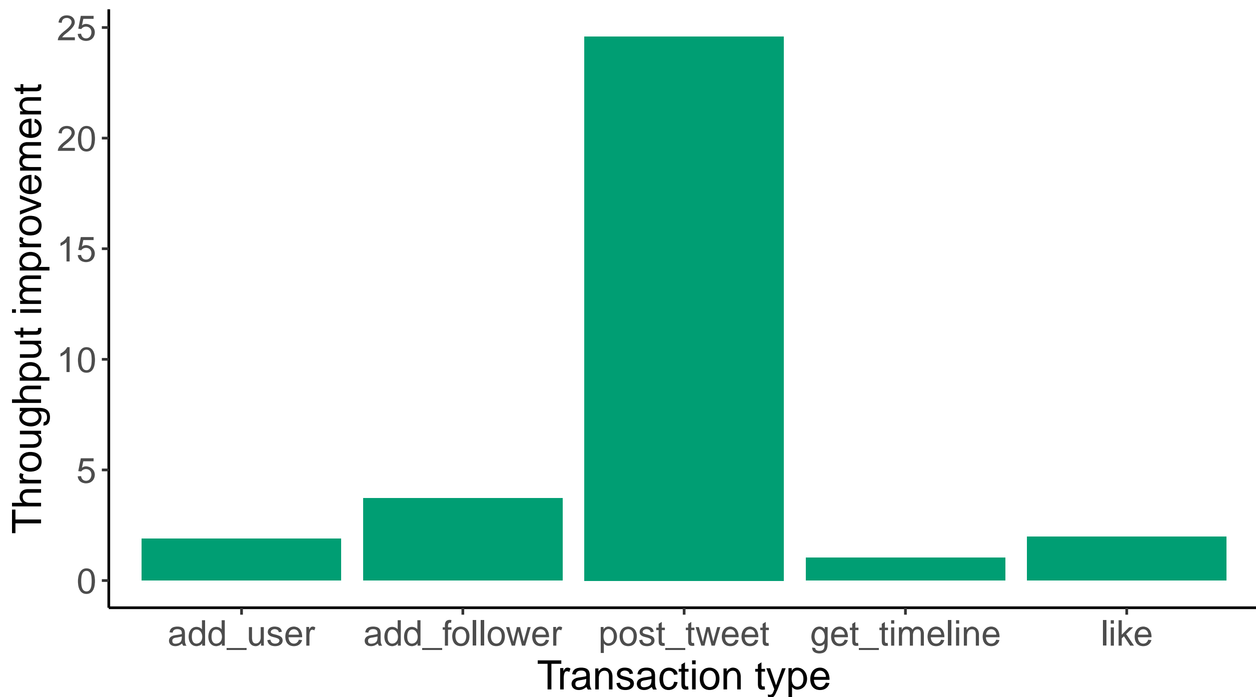

Figure 2.8: Throughput improvement by transaction type with DOCC.

Benefit of data push notifications. Diamond's data push notifications piggyback client cache entries along with notifications, allowing clients to read from their local caches instead of performing wide-area round trips. Two-phase commit ordinarily requires at least one round trip for reads and another round trip to commit, so this optimization is essential to maintaining competitive latencies with built-from-scratch data management solutions that submit client operations in a single message.

To study the effect of data push notifications, we built another benchmark based on the 100 game [104], a simple game in which players take turns adding numbers to a total. Clients use a read-write transaction to take their turn and a reactive transaction to receive other players' turns. This game is ideal for data push notifications, since each client's read set remains the same across turns, and the read sets of the read-write and reactive transactions overlap. We ran pairs of clients that played against each other, taking turns as soon as they could, and measured the latency from one player's client for each player to take one game

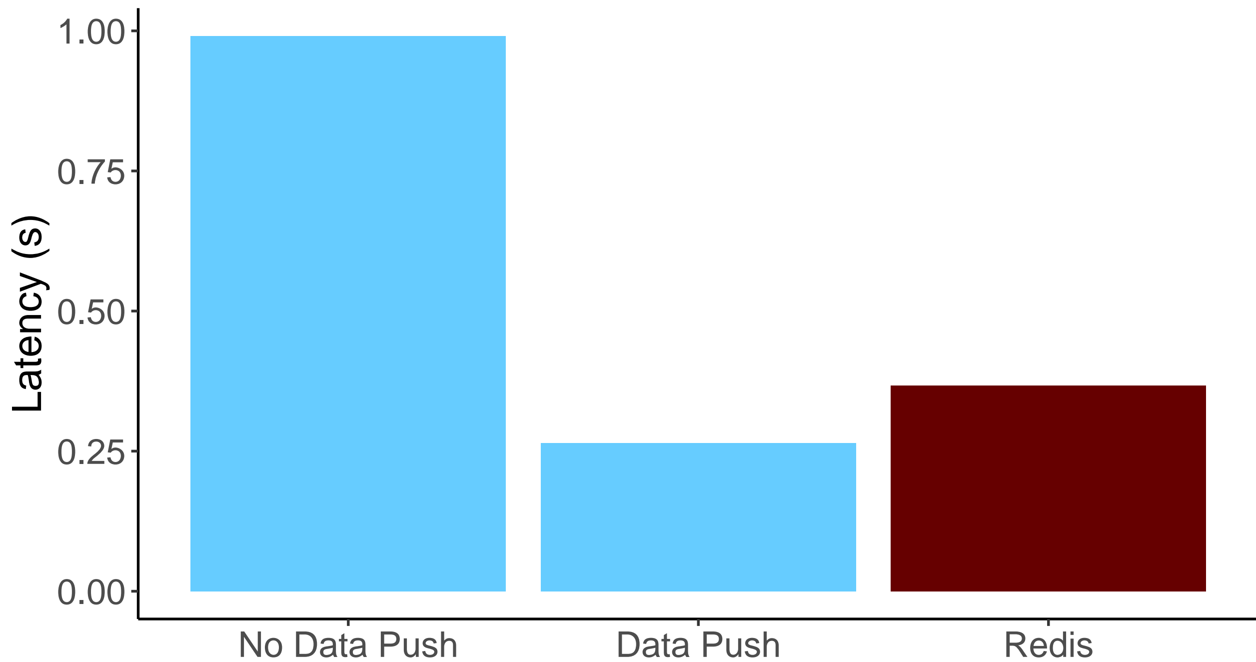

Figure 2.9: Round latencies for our 100 game benchmark with and without data push notifications.

turn (one *round*). We increased the number of client pairs until we saturated the servers and measured latencies at the saturation point. We also built a baseline version that uses Redis as its backend and includes updated game data with its turn notifications.

Figure 2.9 compares Diamond's round latency with and without data push notifications to the baseline Redis game. Without data push notifications, each read-write transaction requires two round trips to complete, and each reactive transaction requires one round trip. With data push notifications, both transactions can read from the client cache, reducing the number of round trips to one for read-write transactions and zero for reactive transactions. As a result, data push notifications reduce the Diamond game's round latency by around 75% and bring Diamond's latency below that of the baseline Redis game, improving performance while providing stronger guarantees.

2.7 Conclusion

Developers of reactive distributed applications have long faced a dilemma: expend significant time and resources building data management infrastructure themselves, or rely on wide-area storage systems with weak guarantees and unwieldy asynchronous interfaces. Diamond solves this dilemma by providing end-to-end data management with strong isolation guarantees, high throughput, and a simple interface. Diamond introduces reactive data types, application-level objects with data structure semantics tied to global shared data items, and the `rmap()` primitive, which developers use to bind reactive data types to shared data. Developers work with reactive data types by using read-write transactions to modify shared data and reactive transactions to listen for updates; both transaction types let developers write synchronous code, eschewing the chains of nested callbacks favored by existing Backend-as-a-Service systems. Diamond provides strong, configurable isolation guarantees for these transactions, and it relies on a set of key performance optimizations to deliver those guarantees with good performance: DOCC enables Diamond to run read-write transactions at strong isolation levels with high throughput, and data push notifications let Diamond execute reactive transactions with minimal wide-area round trips. Our experience building apps with Diamond and retrofitting existing apps to use Diamond showed that Diamond simplifies app code and transparently adds isolation and fault-tolerance guarantees, and our evaluation showed that DOCC improves application throughput and that data push notifications reduce reactive transaction latency.

Chapter 3
HERCULES

This chapter continues our examination of shared data management, changing focus to the end-user experience. Distributed apps need to respond instantly to user input, and the best way for them to do so is to expose uncertainty in shared data, but the interfaces of existing storage systems, including Diamond, make it hard for developers to build this behavior. This chapter presents *Hercules*, a distributed storage system that lets apps expose uncertainty by providing multiple views of shared data. Our Hercules design targets *real-time interactive apps*, wide-area distributed apps with constant interaction between users. Whereas Diamond focused on providing high throughput as app deployments scale up, Hercules focuses instead on delivering low-latency interactions as perceived by end users. This chapter is based on work published as a University of Washington Technical Report [64].

3.1 Introduction

Real-time interactive apps are an important and increasingly prominent class of distributed applications. These apps let users manipulate a set of shared data and interact with each other in real time. They run on modern client devices such as smartphones, tablets, virtual reality (VR) headsets [82], and interactive whiteboards [72], which have natural user interfaces with touch or gesture input and high-resolution visual output. Some examples are collaborative drawing and design apps, multiplayer games, messaging apps, and collaborative office apps like Google Docs.

Three properties distinguish real-time interactive apps from traditional distributed apps:

1. They feature *small groups of users concurrently modifying shared state.* Users continuously respond to each others' actions and interleave operations.

2. They have *tight responsiveness bounds* for updating output in response to input. Human-computer interaction research has established bounds of around 100ms [73, 80], and new devices such as VR headsets have even tighter bounds of 7-20ms [2].

3. Clients are *distributed over the wide area.* Users participating in the same experience may be located in different regions or parts of the world, and even co-located users have client devices separated from cloud infrastructure by high-latency last-mile network connections.

Real-time interactive apps seek to provide the illusion that each user is directly manipulating shared data, but creating this illusion is difficult when multiple users separated by wide-area latency gaps are concurrently interacting with shared data, and each user expects an instantaneous response from their client. If a client instantly updates its output to optimistically reflect a user's operation, it risks "lying" to the user and having to correct its output if the operation ends up conflicting with concurrent operations or failing to reach the backend. This error correction results in a jarring and unsatisfying user experience. On the other hand, if the client waits to update its output until the user's operation has been acknowledged by the backend, it will violate its responsiveness bounds. The best option clients have is to *expose uncertainty*, updating their output immediately in response to user input but visually indicating that the user's operations have only been tentatively applied.

Diamond's interface makes it hard to implement a client that responds instantly to user input and exposes uncertainty. Diamond encourages a simple programming model in which developers define read-write transactions to modify RDTs and define reactive transactions that render the UI based on RDTs. This approach results in long delays between when the user makes an input and when the client updates its output, since there is a wide-area round trip between when a read-write transaction finishes and when the corresponding reactive transaction begins.

In order to have the client update its output instantly and expose uncertainty, developers need to use a more complicated approach. They must maintain separate local state mirroring

the RDTs and tracking uncommitted user actions, use that local state to render the UI, and use reactive transactions to maintain coherence between RDTs and the local state. The challenge only increases if developers want finer-grained information about the progress of operations, such as tracking which operations have been persisted on the client's durable storage or are visible to other clients. This weakness is not unique to Diamond; the same approach is necessary with any distributed storage system that exposes only one version of shared state to the client. Section 3.2 describes the problem in more detail.

This chapter presents *Hercules*, a new distributed storage system that lets clients easily expose uncertainty and update output in response to input. Hercules provides a client-side cache with multiple *views* of shared state, where each view represents a different tradeoff of consistency and staleness. Hercules provides low-latency access to all its views, so developers can use them directly on the UI critical path, and it provides novel state views at the extreme ends of the staleness/consistency continuum, which are useful to apps but unsupported by existing systems.

Like Diamond, Hercules lets developers define application operations and UI logic by writing simple, synchronous code that executes as an atomic unit. Hercules differs from Diamond in that the real-time interactive apps it targets have different workload characteristics from the large-scale distributed apps targeted by Diamond, with smaller user groups sharing smaller amounts of shared state. Hercules takes advantage of this difference in workload to offer stronger semantics, including a view that tracks when operations are visible to other clients.

Our contributions in this chapter are as follows:

- We identify the unique needs of real-time interactive apps, and we identify multi-versioned client-side caching as a means of satisfying those needs.

- We describe the design and implementation of Hercules, a storage system that uses client-side caching to provide low-latency views of shared state, including novel views not supported by existing systems.

- We present a set of example applications that showcase the use of Hercules's different state views, and we evaluate Hercules's performance optimizations and recovery protocols.

We implemented our Hercules prototype and example applications in C# on top of the AMBROSIA reliable RPC framework [42]. Our experiments show that apps built on Hercules have 1.7x–9x lower operation delays than apps built on a commercial cloud storage system, that Hercules's batching maintains stable performance as a deployment scales to include more clients, and that its recovery protocol allows it to maintain liveness in the face of client failures.

3.2 Motivation

This section describes what real-time interactive apps need from a distributed storage system, and it shows how existing storage systems do not meet those needs.

3.2.1 The Needs of Real-Time Interactive Apps

The tight responsiveness bounds and wide-area distribution of real-time interactive apps mean that they need a low-latency client-side cache of shared state. This cache has three requirements, which we describe below.

Simultaneous access to multiple views. Multiple views of shared state let real-time interactive apps easily expose uncertainty when rendering client output. This uncertainty comes from the fact that clients must respond to user input immediately, but that input may conflict with other users' concurrent actions. Apps typically handle this tension by speculatively applying a user's own actions right away and then correcting the output once the storage system orders the actions and resolves conflicts. Some apps indicate which parts of the output are uncertain, but implementing that functionality currently requires substantial developer effort, so many apps simply treat speculative output as authoritative and risk "lying" to the user for brief periods.

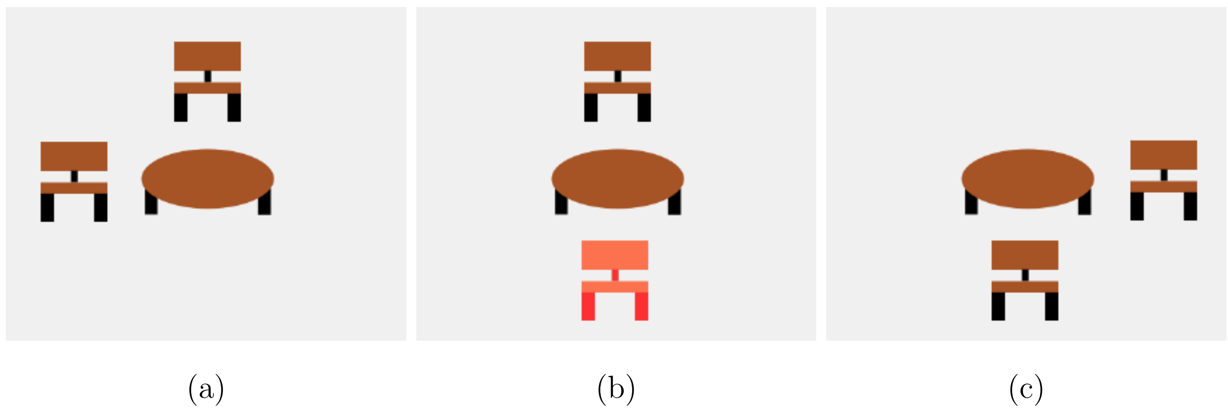

Figure 3.1: A sequence of screenshots from our interior design app, showing how it exposes uncertainty to the user. In (b), the user has just moved one of the chairs, and that change has not yet been synchronized with the cloud or other users, so the client highlights the chair in red.

We implemented an "interior design" app as an example of how an app can indicate uncertainty in its output. The app lets users collaboratively design a room's layout by moving around virtual items of furniture. The interior design app exposes uncertainty by highlighting furniture items whose position is uncertain (i.e., items which the user has just moved), as shown in Figure 3.1.

The easiest way to identify uncertainty is to compare different views of shared state representing different consistency levels. However, most existing systems provide interfaces that expose only a single state view at a time, even if they support multiple consistency levels. In order to simultaneously access multiple state views using these systems, apps must invoke read operations at multiple consistency levels and then manually cache the results.

Views that trade off staleness and consistency. The tight responsiveness bounds of real-time interactive apps mean that they must render each output frame very quickly, within tens of milliseconds. With those short frame times, apps cannot afford to make long-running queries for shared state on the output-rendering critical path.

Existing systems may provide immediate access to weakly consistent state views, but for

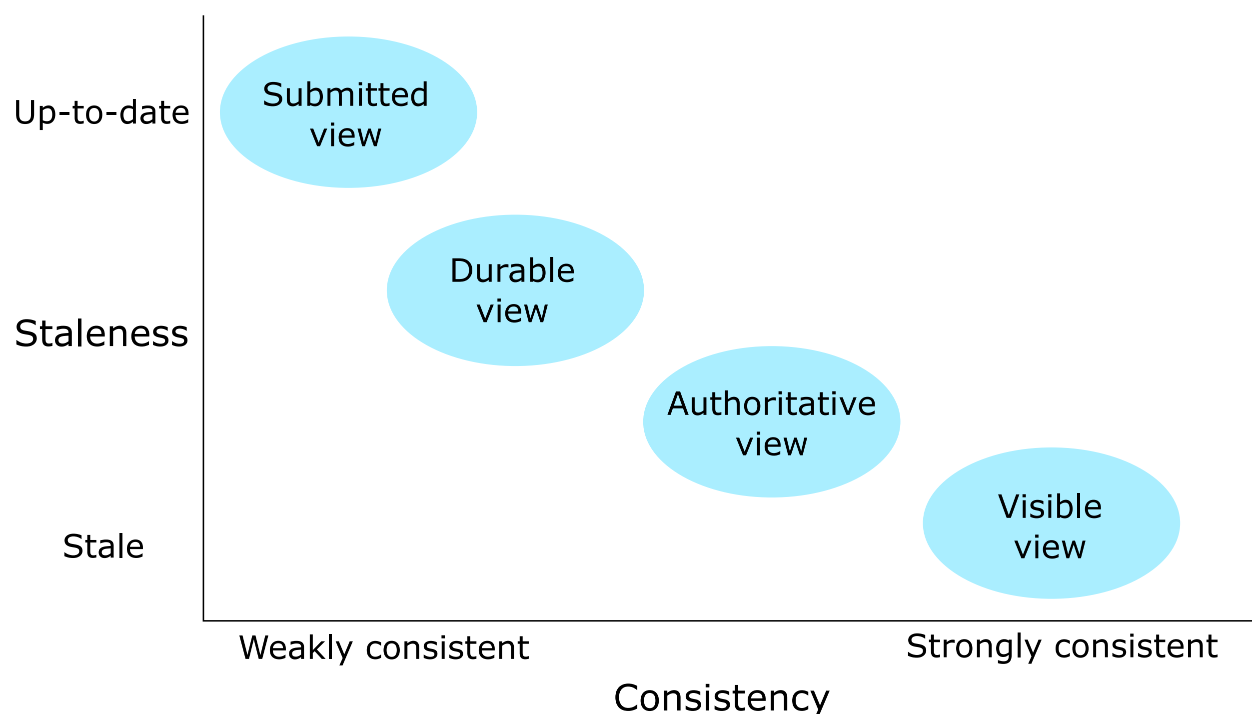

Figure 3.2: An illustration of the relationship between staleness and consistency. A highly available state view cannot be both up-to-date and strongly consistent, so each state view represents a tradeoff between the two properties.

strongly consistent views, they generally require apps to either perform a blocking query or register an asynchronous callback. These systems therefore offer a tradeoff between consistency and performance (access latency). Real-time interactive apps cannot afford to sacrifice access latency when rendering output, but they can tolerate staleness when reading strongly consistent versions of state. They need a storage system that instead exposes the tradeoff between staleness and consistency. Figure 3.2 illustrates this tradeoff.

New, unique views. Real-time interactive apps benefit from views of shared state that are not supported by existing storage systems. Most existing systems provide a strongly consistent, *Authoritative* state view representing the agreed-upon ground-truth version of state at a point in time. Many systems also provide a weakly consistent state view that

includes operations that have been persisted locally on the client but have not yet propagated to cloud storage. We refer to this view as the *Durable* view. These two views alone are insufficient for real-time interactive apps.

Due to their responsiveness needs, real-time interactive apps need a low-latency state view that includes operations before they are locally persisted. Modern immersive client devices such as VR headsets have extremely tight response bounds of 7-20ms for updating output in response to input [2], and those bounds will decrease in the future. Apps running on these devices cannot wait for changes to shared state to be locally persisted before those changes are reflected in their output. They need a view including operations that are only stored in memory, which we call the *Submitted* view.

On the other end of the consistency spectrum, because real-time interactive apps have concurrent user activity, they also benefit from a view including only operations that other users have received. We refer to this view as the *Visible* view. Many popular apps [33, 55] provide such visibility information, but because storage systems do not track operation visibility, apps are left to implement this behavior themselves.

3.2.2 Current Systems

Widely available distributed storage systems do not meet the requirements outlined above. Today's real-time interactive apps must implement the missing functionality themselves, a difficult task made harder by the opaque interfaces of storage systems. This section examines a popular distributed storage system and describes the problems it presents for these apps.

Firebase Cloud Firestore [44] is a representative example of a distributed storage system. It provides a data store organized into collections of JSON documents. Cloud Firestore allows client apps to submit writes or transactions modifying shared data, and it supports event listeners that invoke a callback when a document or query set changes. An event listener passes the most recent snapshot of shared data into its callback. Cloud Firestore optimistically applies a client's own operations locally before sending them to the backend, and it invokes event listener callbacks in response to those optimistic invocations. Event

listeners provide metadata that can be used to distinguish between notifications for local and backend events.

Apps must explicitly store operations. An app that maintains multiple views of shared state must model and store app-level operations in memory, and it must implement protocols to re-compute its views using those operations when Cloud Firestore notifies it of changes to shared state. If the app wants to use views that lag behind the most recent version of data in Cloud Firestore, such as the *Visible* view described above, it must also store those operations *in Cloud Firestore* rather than simply storing the shared state itself.

Notifications are too coarse-grained. Cloud Firestore's event listeners can register for changes to either a single document or the results of a query. With only these two options, apps cannot subscribe just to notifications for new operations. They must instead listen to the entire set of operations, figure out which operations are new, and act on those operations. This requirement both adds an extra programming burden (developers must implement de-duplication logic) and reduces performance (the entire set of operations is delivered every time a new operation is added).

Apps must perform manual synchronization. Cloud Firestore's event listener callbacks execute on background threads, while real-time interactive apps typically process input and render output on a dedicated, non-blocking user interface thread. Apps must therefore add synchronization logic to ensure that event listener callbacks do not access shared state or operation metadata at the same time as input-processing or output-rendering methods, and they must structure the event listener callbacks to execute without blocking, so that the added synchronization does not cause the user interface to block.

3.3 Hercules Overview

Hercules is a distributed storage system that exposes different views of shared state to clients, where each view reflects a particular tradeoff of consistency and staleness.

3.3.1 System Model

Hercules targets distributed applications with a client/server architecture, in which users interact with client apps that submit operations to a central server, and the server maintains the authoritative copy of shared state and fans out operations to other clients. Although our current Hercules prototype supports a single (possibly replicated) server, Hercules could be extended to support more sophisticated backend architectures by modifying or adding new state views.

Hercules targets real-time interactive apps, where small numbers of users (e.g., tens of users) continuously modify shared state and respond to each others' operations. Hercules does not target distributed apps with different workloads, such as social media services that have millions of users and take minutes to fully propagate operations. Many of Hercules's techniques could be applied to these apps in a straightforward manner, but some would require adjustment (for instance, it might be impractical to track which of a user's operations are visible to all other users).

3.3.2 Programming Interface

Hercules models an application's shared state as a state machine. The state machine starts at some initial state and moves between states according to the operations in a totally-ordered operation log. Hercules supports two kinds of operations: *read-write* operations and *read-only* operations. Read-write operations can perform arbitrary deterministic computation, but they cannot return values or externalize state, because they are applied speculatively to some views. Read-only operations read a particular view of the state machine. Read-write operations are stored in the operation log, while read-only operations are unlogged.

When using Hercules, the application developer writes a state machine class whose variables define the shared state and whose methods define the read-write operations. Hercules generates a client library with two kinds of methods: operation stubs matching the read-write operations, and a `GetState()` method that returns the requested view of shared state.

Applications perform read-write operations by calling the operation stubs, and they perform read-only operations by calling `GetState()` and then directly reading from the returned view.

3.3.3 Hercules State Views

The Hercules client library provides different *views* of the shared state, where each view is a read-only copy of the state machine resulting from a different operation log. The different views trade off consistency and staleness. In this context, staleness describes how long a client must wait for its own operations to be included in a view; a weakly consistent view allows operations to be inserted into the middle of its log, while a strongly consistent view only permits operations to be appended to the end of its log. Our Hercules design exposes four views, in order of increasing staleness:

- The *Submitted* view is a weakly consistent view including all operations that have been submitted by this client.

- The *Durable* view is a weakly consistent view that guarantees that its operations will eventually become visible to all clients.

- The *Authoritative* view is a strongly consistent view that guarantees that its operations will eventually become visible to all clients.

- The *Visible* view is a strongly consistent view that guarantees that all of the client's own operations contained in it are visible to all clients.

Each view is defined in terms of a *global operation log*, which is agreed upon by all clients and which specifies the "master copy" of shared state. Operations submitted by clients eventually enter the global log; how exactly clients agree upon the global log depends on the precise system architecture. A client's *Authoritative* log is a prefix of the global log, and its *Visible* log is a prefix of the *Authoritative* log. The *Submitted* log consists of the *Authoritative*

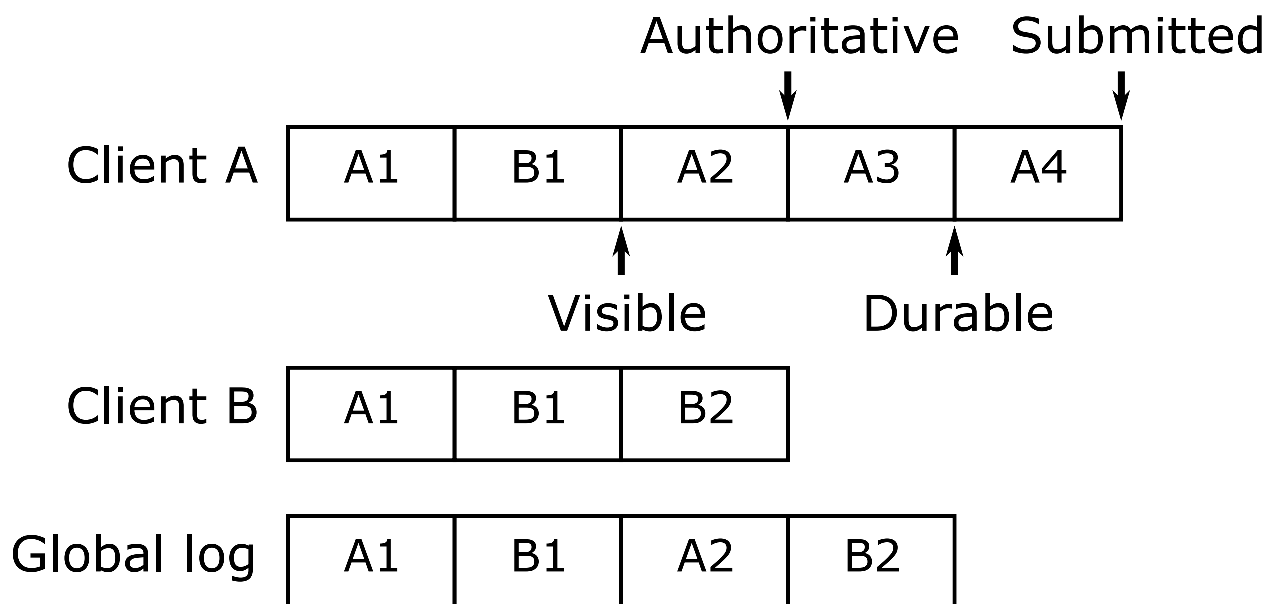

Figure 3.3: An example of the relationship between a client's state views and Hercules operation logs. Client A's operations A1 and A2 are in the global log, but only A1 is visible to client B. Its operation A3 is guaranteed to eventually make it into the global log, but no such guarantee applies to A4.

log with all of the client's submitted operations appended to the end, and the *Durable* log is the *Authoritative* log with a prefix of the client's submitted operations appended. Figure 3.3 illustrates these definitions with an example.

These definitions result in a clean set of prefix relationships between the views, where each view's operation log is the prefix of another view's. The client's *Visible* log is a prefix of its *Authoritative* log, which is a prefix of its *Durable* log, which is a prefix of its *Submitted* log.

Note that our definition of the *Visible* view is concerned with the visibility of a client's own operations. Operations from other clients that are not yet visible to everyone may appear in it. We found that this definition results in a view that is intuitive to use, since users are often primarily interested in whether their own operations are visible to others. However, there may be situations where (for instance) Alice sees Bob's operation and wants to know whether Carol has also seen that operation. In that case, a view that shows operations from any client only if they are visible to all clients would be useful. Such a view would fit cleanly into Hercules's design but would require more messages per operation.

3.4 Hercules Design

Hercules provides clients with views of shared state that represent different tradeoffs of consistency and staleness. In the context of a concrete storage system, another interpretation of the views is that each view contains operations that have passed beyond a certain "radius" in the system, from the point of view of the client. To provide those different views, Hercules turns each operation into multiple RPCs that provide finer-grained information about the operation's progress through the system. The Hercules client library uses those RPCs to derive its views.

3.4.1 AMBROSIA Background

Hercules is built on top of AMBROSIA [42], a distributed runtime that provides reliable execution and in-order, exactly-once RPC delivery across nodes. We used AMBROSIA to

build Hercules because its reliability guarantees simplify Hercules's design.

AMBROSIA nodes are called *Immortals*; each Immortal consists of a package of persistent state and a set of deterministic RPC handlers that operate on that state. An Immortal can call RPCs on other Immortals or perform self-calls of its own RPCs. A special type of RPC is an *Impulse Handler*, which an Immortal calls on itself to accept non-deterministic external input.

3.4.2 System Architecture

A Hercules deployment consists of a server program and one or more client programs running on end-user devices. Hercules generates the server program and a client library; app developers write code for a client program that accepts user input and renders output using the client library. The server stores the global operation log. To save space, it maintains a checkpoint copy of shared state that represents a prefix of the global log and only explicitly stores the suffix of the log. The server stores all of its state in durable storage, whereas the client splits its state across durable storage and volatile memory. Figure 3.4 illustrates this architecture.

The Hercules server program runs an AMBROSIA Immortal, and each instance of the Hercules client library runs an Immortal on a background thread. All communication between the server and client library happens via RPCs between their Immortals, and each component stores its persistent state inside of its Immortal.

In the context of Hercules's system architecture, each view has the following interpretation:

- The *Submitted* view includes all operations that have been submitted by this client, even those that are only stored in memory.

- The *Durable* view includes submitted operations from this client that have been durably written to the client's local storage.

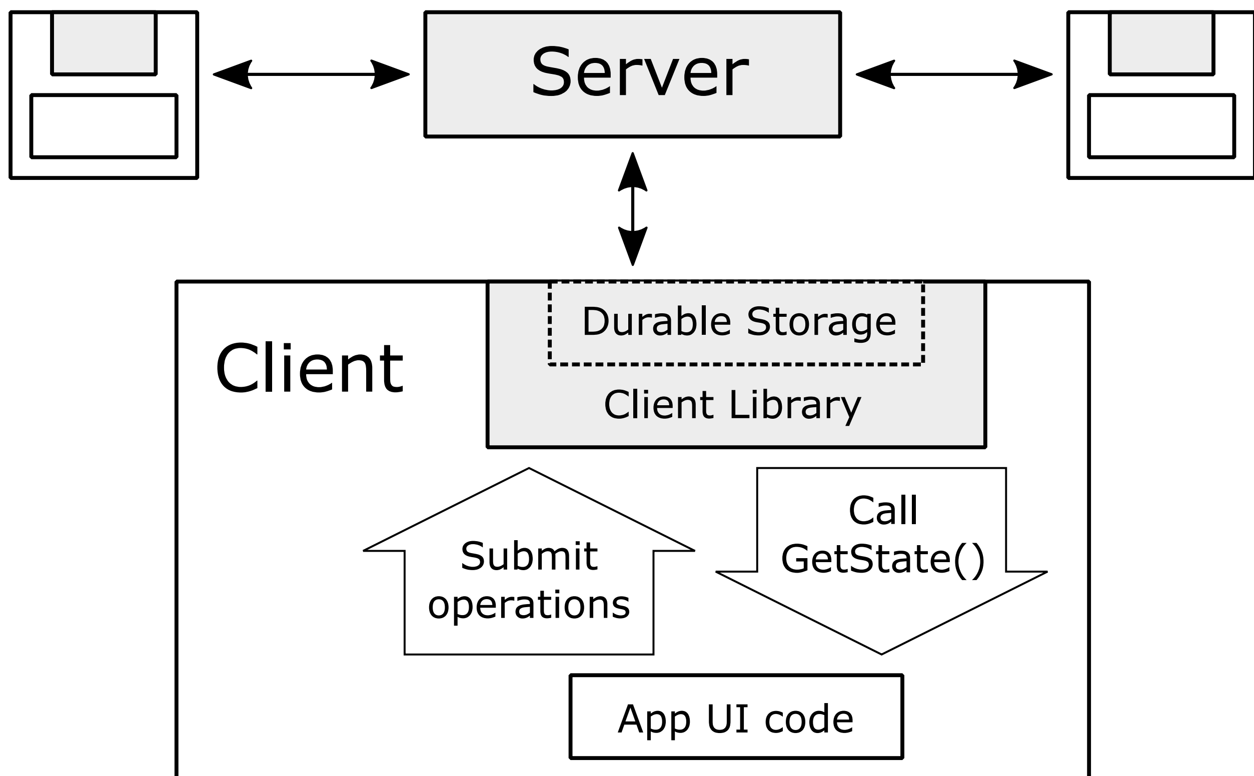

Figure 3.4: The architecture of a Hercules application. Shaded areas represent components provided by Hercules.

- The *Authoritative* view includes operations that have been accepted and ordered by the server.
- The *Visible* view includes all operations up to (but not including) the first operation from this client that has not been delivered to all other clients.

Operations in the *Submitted* view may be lost if the client app crashes, and operations in the *Durable* view may be lost if the client device suffers a persistent storage failure.

3.4.3 Operation RPC Protocol

When a client app performs an operation, Hercules transforms that operation into multiple RPCs that propagate it through the system and inform the client library of its progress. The operation propagation protocol, illustrated in Figure 3.5, has the following steps:

1. The client app calls an operation stub exposed by the Hercules client library. Once the stub call returns, the operation is *Submitted*.
2. The client library calls the *Make-Durable* ImpulseHandler RPC on itself to locally persist the operation. The operation is now *Durable*.
3. The client library sends a *Deliver-Operation* RPC containing the operation to the Hercules server.
4. The server adds the operation to its global operation log and sends a *Notify-Auth* RPC to the client library. The operation is now *Authoritative*.
5. The server sends *Remote-Operation* RPCs containing the operation to the other clients.
6. Each other client responds with an *Remote-Ack* RPC.
7. Once the server has received acks from all other clients, it sends a *Notify-Visible* RPC to the client library. The operation is now *Visible*.

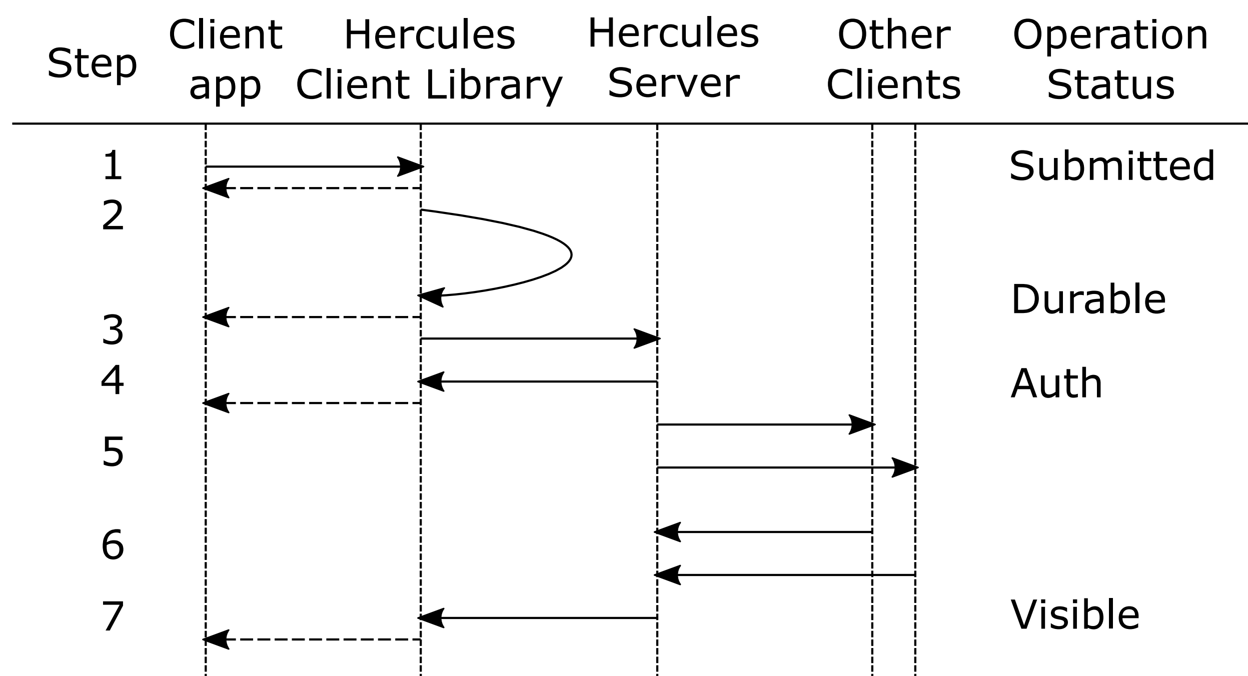

Figure 3.5: The Hercules operation protocol. Dashed lines represent the client app reading updated state views from the client library by calling the `GetState()` method.

Each RPC call is asynchronous and non-blocking. At any time, the client app can call the `GetState()` method to get the client library's current state views.

3.4.4 Client View Computation

The Hercules client library uses the RPCs it receives to assemble its state views. This procedure requires some care. This section describes the basic algorithm for computing state views; Section 3.4.5 describes a version optimized for high-throughput scenarios.

The client library maintains a persistent copy of state corresponding to its *Visible* view and transient copies for the other views, and it saves computation by directly applying operations to those transient copies when possible rather than re-deriving them from scratch. Furthermore, because the full operation log of a given state view is a prefix of the log of its less-stale counterpart (e.g., the *Authoritative* log is a prefix of the *Durable* log), the client

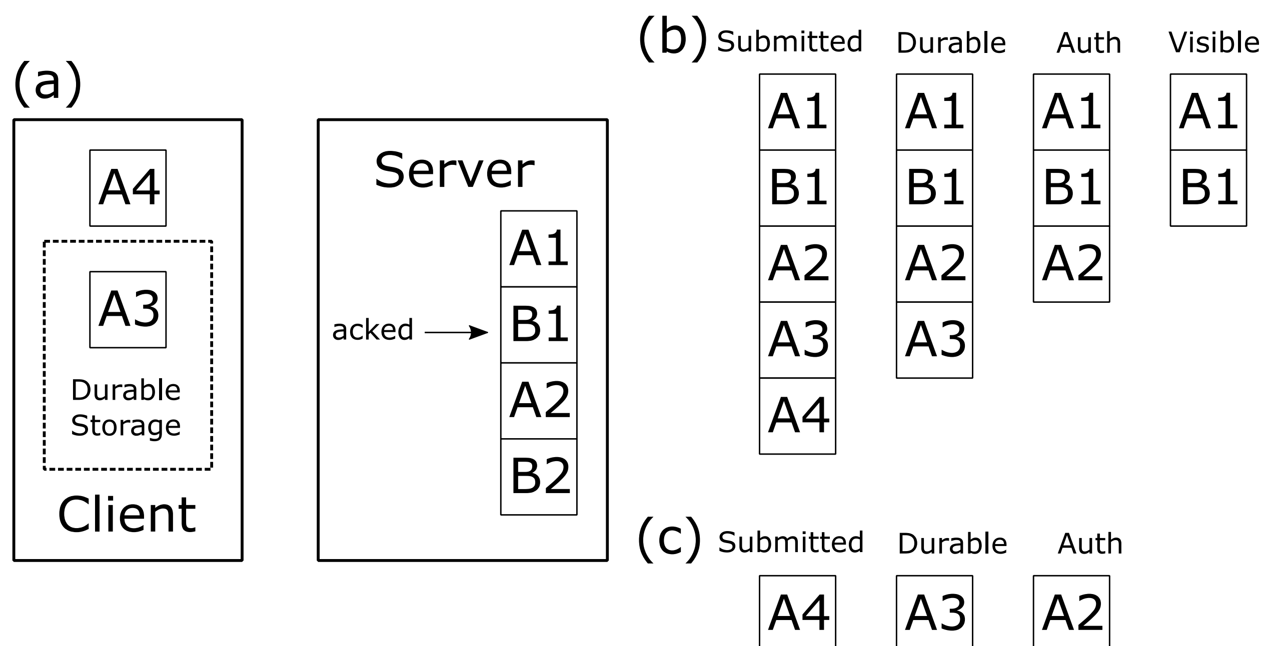

Figure 3.6: An example of the correspondence between (a) the progress of operations through Hercules, (b) the operation logs reflected in the client library's state views, and (c) its operation lists used to compute those state views.

library saves space by storing *operation lists* that contain only the operations that are part of one log and not its less-stale counterpart (e.g., the *Durable* list holds the operations that have been persisted but have not yet been received by the server). It uses these operation lists to re-derive state views when necessary. Figure 3.6 shows the relationship between the operation logs and operation lists.

The client library updates its state views whenever it receives an RPC from the server (summarized in Figure 3.7). When the RPC notifies it about the progress of one of its own operations, then the operation log for one of its state views grows to be a larger prefix of its less-stale counterpart, but the operation logs for the other state views are all unchanged. In this situation, the client library can directly apply the operation to the affected view without changing any of the others.

However, if the RPC contains a remote operation from another client, then the remote

Component	RPC	Caller	Description
Client	Make-Durable	Client	Persist a locally submitted operation.
	Notify-Auth	Server	Alert the client that its operation is stored on the server.
	Notify-Visible	Server	Alert the client that its operation is visible to all clients.
	Remote-Operation	Server	Deliver an operation from another client.
Server	Deliver-Operation	Client	Deliver an operation from the calling client.
	Remote-Ack	Client	Acknowledge an operation from another client.

Figure 3.7: The Hercules client and server RPC interfaces.

operation will be inserted into the middle of several views' operation logs. Whenever an operation is inserted into the middle of an operation log, the client library must re-compute the corresponding view by starting with a copy of its more-stale counterpart and re-applying the subsequent operations in the log. We call this re-computation a *rebase*. We can perform rebasing more efficiently by rebasing multiple views at once (e.g., rebasing the *Durable* view off of the *Authoritative* view, and then rebasing the *Submitted* view off of the *Durable* view).

Specifically, the client library takes the following actions when it receives RPCs corresponding to some operation O:

- *Make-Durable*: The *Durable* view changes. The client library removes O from its *Submitted* list, appends O to its *Durable* list, and directly applies O to the *Durable* view.

- *Notify-Auth*: The *Authoritative* view changes. The client library removes O from its *Durable* list, appends O to its *Authoritative* list, and directly applies O to the *Authoritative* view.

- *Notify-Visible*: The *Visible* view changes. The client library removes O and any preceding operations from its *Authoritative* list and directly applies those operations to

its *Visible* view in order.

- *Remote-Operation*:
  - If any of this client's *Authoritative* operations are not yet *Visible*: the *Authoritative*, *Durable*, and *Submitted* views change. The client library appends O to its *Authoritative* list, directly applies O to the *Authoritative* view, and rebases the *Durable* and *Submitted* views.
  - If all of this client's *Authoritative* operations are *Visible*: all four views change. The client library directly applies O to the *Visible* view and rebases the other three views.

3.4.5 Client Rebase Batching

In the client view computation algorithm introduced in Section 3.4.4, a client performs a rebase whenever it receives a *Remote-Operation* RPC. Each rebase involves re-executing multiple operations, so rebasing can become prohibitively expensive when other clients are performing operations frequently, or when high latencies between the clients and the server result in large *Durable* or *Auth* lists. As a result, the Hercules client supports a batched rebase mode which performs rebases at a coarser time granularity.

When rebase batching is enabled, the client reacts to *Notify-Auth*, *Notify-Visible*, and *Remote-Operation* RPCs by enqueuing them in a *batch list* instead of performing any of the actions described in Section 3.4.4. At an app-defined time interval (e.g., every 200ms), the client dequeues each of the RPCs in its batch list and modifies its operation lists in response to each one, but it does not modify state views or perform any rebasing. After it is done processing the batched RPCs, the client performs a single rebase operation to refresh its state views. Finally, it sends *Remote-Ack* RPCs for any remote operations that were in the batch.

Rebase batching lowers the rate at which the client refreshes its state views to reflect

remote operations. In exchange, it lowers the execution burden on the client in scenarios of high system throughput. Whether an individual application should enable rebase batching, and the precise batching interval it should use, depends on the characteristics of the specific application and its deployment.

3.4.6 Recovery from Failures

Hercules's state views provide a range of reliability guarantees. As a result, it is important for Hercules to respond to failures correctly, recovering the reliable views and cleaning up the unreliable views. Hercules also needs to track which clients are active in order to prevent the *Visible* view from being blocked by failed clients. AMBROSIA's reliability guarantees simplify these tasks, but Hercules's design requires care in choosing when and how to use those guarantees.

Recovering state views. Of the four state views exposed by Hercules, the *Submitted* and *Durable* views are potentially affected by client failures.

Hercules makes no guarantees about the *Submitted* view; its operations may be lost at any time due to transient failures. It is this property that enables Hercules to quickly update the *Submitted* view in response to user input. The client library stores the *Submitted* operation list in volatile memory and adds submitted operations to it inside of a regular method rather than an AMBROSIA RPC. If the client program dies and restarts, it simply clears the *Submitted* operation list during recovery.

In contrast, Hercules guarantees that operations in the *Durable* view will eventually be sent to the server, unless the client suffers a persistent storage failure. It maintains this promise by including the *Durable* operation list in the client Immortal's persistent state and only appending to it inside of an AMBROSIA RPC. If the client app suffers a transient failure, its recovery procedure restores its *Durable* operation list, and AMBROSIA guarantees that its *Deliver-Operation* RPC is eventually sent to the server. On the other hand, if the client device's persistent storage fails, then the operations in its *Durable* operation list are lost. In

this situation, the client app would start up as a new client instance, and any operations that it successfully delivered to the server before failing would be reflected in its *Authoritative* and *Visible* views.

The server stores its global operation log and visibility-tracking metadata in its Immortal's persistent state, and it performs all modifications to that data inside of AMBROSIA RPCs, making it robust to transient failures. The server can use replication or another method to avoid persistent storage failures; such a failure would require the Hercules instance to start over from scratch.

Maintaining the visibility set. In addition to maintaining its guarantees in the face of failures, Hercules must also ensure that it remains useful when clients crash or disconnect. This concern primarily affects the *Visible* view, which progresses only when *all* clients have received a client's operations. In practice, defining the *Visible* view in terms of all clients is impractical when some of those clients may experience transient connection issues or failures. As a result, Hercules defines a *visibility set* containing the clients that are actively receiving operations. It guarantees that a client's operations in the *Visible* view have been delivered to all clients in the visibility set, and it notifies clients whenever the visibility set changes.

The Hercules server detects crashed or disconnected clients by periodically checking for clients with acks that have been outstanding for more than some timeout interval (e.g., every two seconds). Once the server detects a timeout, it updates its visibility set and sends out a *Visibility-Set-Change* RPC to every other client. It then checks every operation that is not yet visible; if it has been acked by every client except the just-removed client, the server marks it as visible and sends a *Notify-Visible* RPC to its sender.

The server must also alert the client that it has been removed from the visibility set (if the client is alive but suffering from connection issues) and provide it with a way of re-joining after it recovers. AMBROSIA's reliability guarantees mean that a crashed client is indistinguishable from a client with very long RPC delays, so Hercules handles crashed and disconnected clients with the same protocol.

When the server removes a client from the visibility set, it sends the client a *Deregister* RPC. AMBROSIA guarantees that the client will eventually receive the RPC after it restarts or its connection improves. Upon receiving the RPC (after it has recovered), the client immediately sends the server a *Register* RPC. The server then adds the recovering client to the visibility set, broadcasting another *Visibility-Set-Change* RPC, and "catches up" the client by sending it the data required to reconstruct its views.

In order for the server to provide a recovering client with the data it needs to reconstruct its views, the server has to store the right data internally. As mentioned in Section 3.4, the server explicitly stores a suffix of the global operation log along with a checkpoint copy of shared state representing the prefix of the log. We select the checkpoint prefix to be the prefix containing only operations that are visible to all clients. (Note that this prefix does not necessarily correspond to the *Visible* view of any client; also, note that one or more operations in the server's suffix list may be visible to all clients, even though the first operation in the suffix list is not.)

To "catch up" a recovering client, the server sends the client its copy of state, its suffix list of operations, and the sequence number of the latest operation from the client that is visible. The client uses these items to reconstruct its *Visible* copy of state and its *Authoritative* and *Durable* operation lists.

3.5 Implementation

We implemented Hercules in C# on top of AMBROSIA [42]. Our Hercules prototype includes the core client library and server logic, profiling code to measure their performance, and a code generation program that generates the client library and server interfaces for a particular app. We also implemented a set of example apps used to demonstrate and evaluate Hercules. Our implementation consists of 3269 lines of C# code (1982 LOC for Hercules itself and 1287 LOC for the example apps), not counting any files that are generated by either Hercules or AMBROSIA.

Our client library implementation differs very slightly from the description in Section 3.4,

in that it performs a full rebase of all three state views upon receiving a *Remote-Operation* RPC, rather than rebasing only the *Durable* and *Submitted* views when possible. This difference makes the code simpler at the expense of performing a bit of unnecessary work.

3.6 Application Case Studies

In this section, we describe some of the example apps that we built, in order to demonstrate the benefit of Hercules's state views abstraction as well as to highlight the variation in the ways different apps use that abstraction.

3.6.1 Interior Design App

The interior design app allows users to design the layout of a two-dimensional virtual room. Users click and drag items of furniture, such as tables and chairs, to move them around in the room.

The interior design app client uses the *Submitted* and *Visible* views. It indicates to the user if the movement of an item is not yet visible to other users by highlighting the item on-screen. Once the item's updated position is reflected in the *Visible* view, the client changes the item's appearance back to normal. Figure 3.1 illustrates the client's behavior.

Figure 3.8 shows the interior design app client's code for rendering output. Items in the interior design app can exist at any point in the two-dimensional room, and it is an item's position itself that changes as a result of user actions, so the interior design app uses explicit item IDs to identify differences between the state views. The client iterates over all item IDs and checks if the position of the item with that ID differs between the two state views. If so, it highlights the item when drawing it.

Our implementation of the interior design app uses the *Visible* view to show users the ground-truth version of shared state, but the right choice of view may differ for different deployment scenarios. We assume that users tend to concurrently move around the same items, in which case they benefit from knowing which of their operations are visible to others. If users instead tend to move around separate groups of items, the *Authoritative* view may

```
SharedState submittedState = GetState(SUBMITTED);
SharedState visibleState = GetState(VISIBLE);

foreach (int id in submittedState.Items.Keys) {
    Item submittedItem = submittedState.Items[id];
    Item visibleItem = visibleState.Items[id];

    bool highlight =
            submittedItem.XPos != visibleItem.XPos
         || submittedItem.YPos != visibleItem.YPos;

    DrawItem(submittedItem, highlight);
}
```

Figure 3.8: The interior design app's output-rendering code.

be more appropriate, shortening the latency window in which they see uncertainty. Or, if users work with separate items and use the interior design app on reliable workstations, the *Durable* view could even be suitable. Thanks to Hercules's interface, customizing the view used would require modifying only a single line of code.

3.6.2 Chat App

The chat app provides a shared chat room. Users send messages that are appended to the log, and clients display the most recent messages in the log. Figure 3.9 shows a screenshot of the chat app client, and Figure 3.10 shows the client's code for rendering output.

The chat app uses all four state views. When displaying messages that have been added to the *Durable*, *Authoritative*, or *Visible* views, the client draws an icon alongside the message to indicate its status.

The chat app takes advantage of the log structure of its shared state to identify differences between the views. It renders the ten most recent messages in the *Submitted* view's chat

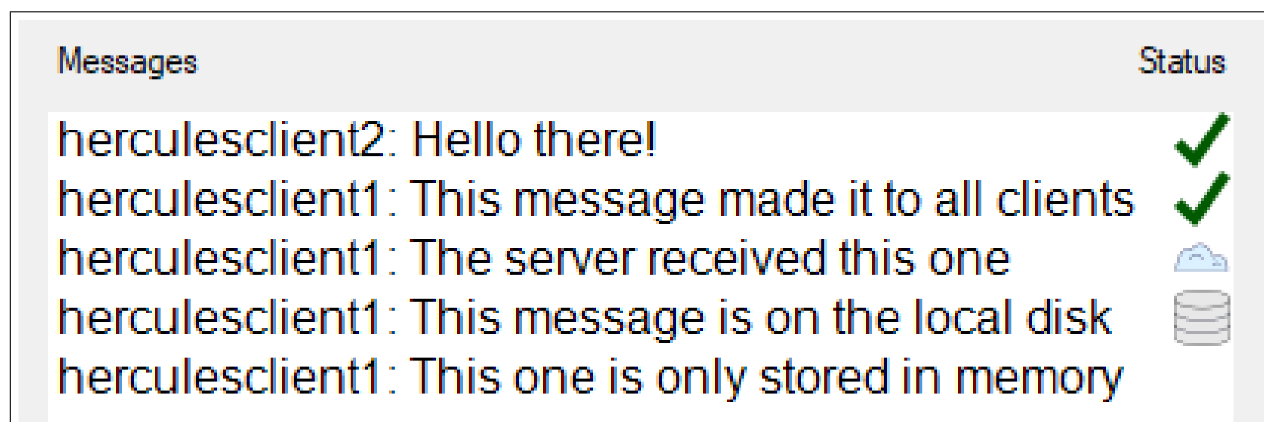

Figure 3.9: A screenshot of the chat app client, showing the different possible message statuses.

log. If those messages are present in the other views, then they will be located at the same index of the chat log in those views. As a result, the client identifies the index of the first message to display, checks for any messages at or after that index in the *Visible* view, and renders those messages with a "visible" icon next to them. The client then moves onto the *Authoritative* view, starting at the index where it left off, and repeats the same process, before moving onto the *Durable* and *Submitted* views.

3.6.3 Spreadsheet App

The spreadsheet app provides a shared spreadsheet that users can collaboratively edit. Each user has a cursor, which they move with the arrow keys, and their edits affect the cell underneath their cursor. The client shows the position of the user's cursor by drawing the selected cell with a red border, and it shows the position of other users' cursors with a blue border.

The spreadsheet app uses the *Submitted* and *Visible* views. The client indicates to the user when their modifications to a cell are not yet visible to other users, by drawing the cell's text in red, and when their cursor movement is not yet visible, by drawing the old cursor position with a thinner border alongside of the updated cursor. Figure 3.11 shows

```
submittedMsgs = GetState(SUBMITTED).Messages;
durableMsgs = GetState(DURABLE).Messages;
authMsgs = GetState(AUTHORITATIVE).Messages;
visibleMsgs = GetState(VISIBLE).Messages;

// Display the 10 most recent messages
int i = submittedMsgs.Count - 10;
for (; i < visibleMsgs.Count; i++) {
    DrawMessage(visibleMsgs[i], /* message */
                i - firstIndex, /* position */
                VISIBLE);       /* status */
}
for (; i < authMsgs.Count; i++) {
    DrawMessage(authMsgs[i],
                i - firstIndex,
                AUTHORITATIVE);
}
for (; i < durableMsgs.Count; i++) {
    DrawMessage(durableMsgs[i],
                i - firstIndex,
                DURABLE);
}
for (; i < submittedMsgs.Count; i++) {
    DrawMessage(submittedMsgs[i],
                i - firstIndex,
                SUBMITTED);
}
```

Figure 3.10: The chat app's output-rendering code.

Part	Qty	Price
Gear		
Bolt		

(a)

Part	Qty	Price
Gear	4	
Bolt		

(b)

Part	Qty	Price
Gear	4	
Bolt	2	

(c)

Figure 3.11: A sequence of screenshots from the spreadsheet app client as two users edit adjacent cells. In (b), the user has entered a value in the "Qty" column and moved the cursor to the "Price" column, but those actions are not yet visible to other users. (Colors edited for better greyscale contrast.)

this behavior. Like the interior design app, the specific views used by this app can be easily customized.

Figure 3.12 shows the spreadsheet app client's code for drawing the display. The spreadsheet app takes advantage of the two-dimensional tabular structure of the shared state when identifying differences between views. The client iterates over each cell in the spreadsheet and figures out whether the cell is selected by its user in the *Submitted* view, by its user in the *Visible* view, or by another user, and draws the cell border appropriately. To determine how to render the cell text, the client simply checks if the cell contents differ between the two views.

Cloud Firestore version. We built a version of the spreadsheet app on top of Firebase Cloud Firestore [44] to compare the Hercules programming experience with that of a popular distributed storage system. Our Cloud Firestore version of the spreadsheet client app required 2.4x as many lines of code as the Hercules version (718 LOC vs. 302 LOC). The Cloud Firestore version provides the equivalent of the *Submitted*, *Authoritative*, and *Visible* views. The Cloud Firestore C# library does not provide the optimistic local invocation feature described in Section 3.2, so we did not implement the *Durable* view.

Most of the additional lines of code implement protocols for maintaining the *Submitted* and *Visible* views. The Cloud Firestore version defines in-memory representations of shared state and operations, maintains a list of submitted operations, and reads and writes an authoritative operation list and a set of per-operation visibility records in Cloud Firestore. It uses event listener callbacks to recompute its views of shared state, mark operations as visible, and update its submitted operation list when its operations are acknowledged by the backend.

Although our Cloud Firestore app is already much larger than the Hercules app, it is missing several features that Hercules provides for free. It does not garbage-collect old operations and visibility records, it does not have the ability to recover from a client failure, and it requires users to manually join and leave the visibility set. Furthermore, it is brittle and

```
SharedState submittedState = GetState(SUBMITTED);
SharedState visState = GetState(VISIBLE);
int numRows = submittedState.Cells.Length;
int numCols = submittedState.Cells[0].Length;

for (int i = 0; i < numRows; i++) {
    for (int j = 0; j < numCols; j++) {
        // Decide color and width of cell border
        Pen borderPen = Pens.Black;
        if (SelectedByMe(submittedState,i,j)) {
            borderPen = new Pen(Color.Red, 4);
        } else if (SelectedByMe(visState,i,j)) {
            borderPen = new Pen(Color.Red, 2);
        } else if (SelectedByOther(visState,i,j)){
            borderPen = new Pen(Color.Blue, 4);
        }

        // Decide color of cell text
        Brush textBrush = Brushes.Black;
        if (submittedState.Cells[i][j]
                != visState.Cells[i][j]) {
            textBrush = Brushes.Red;
        }

        DrawCell(borderPen, textBrush, i, j,
                 submittedState.Cells[i][j]);
    }
}
```

Figure 3.12: The spreadsheet app's output-rendering code.

less efficient than the Hercules version. Cloud Firestore claims to only support one update per second for a given document [43], and based on our experience, the backend enforces that limit by arbitrarily dropping updates when updates are too frequent. To avoid losing writes to the document holding visibility records, we changed the client to mark operations as visible in batches every two seconds. This change makes the *Visible* view slower, and it also does not scale well—we tested it with two clients, but a higher number of clients would require an even longer batching period.

3.7 Evaluation

Our evaluation showed that Hercules's baseline performance exceeds that of comparable apps built on standard cloud storage systems, that Hercules's client-side batching enables it to deliver good performance in high-throughput scenarios, and that its recovery protocol allows the system to progress in the presence of client failures.

For many of our experiments, we used a benchmark application in which the shared state is a large byte array, and clients invoke an operation that increments elements of the array. Each client runs an open loop in which it invokes the operation, calls `GetState()`, and then sleeps for some amount of time. The size of the byte array, number of array increments performed by the operation, and loop sleep interval are all configurable parameters.

3.7.1 Setup

We ran our quantitative experiments on Google Compute Engine virtual machines. We used an n1-highcpu-4 VM (4 virtual cores, 3.6 GB RAM) located in the us-west1 region and an n1-highcpu-8 VM (8 virtual cores, 7.2 GB RAM) located in the us-east1 region. Each VM ran Windows Server 2019. The round-trip latency between the two VMs was 67ms. Unless otherwise noted, we ran Hercules clients on the n1-highcpu-8 VM and the server on the n1-highcpu-4 VM.

3.7.2 Baseline Performance

We compared the normal-case performance of our spreadsheet app to the Cloud Firestore version. For each version of the app, we ran two clients, and on one of those clients, we performed a series of inputs simulating normal user behavior, in which we entered text in every other row in two columns. We measured the delay between when the client submitted each operation and when the operation was reflected in its state views.

We located our Cloud Firestore instance in the same Google Cloud region as our Hercules server. There was no option to locate the Cloud Firestore instance in the us-west1 region, so for this experiment, we put the Cloud Firestore instance in the us-east1 region, and we swapped the VMs so that the n1-highcpu-4 VM ran all clients and the n1-highcpu-8 VM ran the Hercules server. We used Remote Desktop to control the client apps.

Figure 3.13 shows the average delay for the Hercules and Cloud Firestore versions of the spreadsheet app, with error bars indicating the standard deviation. Statistics were computed after discarding the first and last 25% of operations. The figure only reports *Authoritative* and *Visible* delays, because the Cloud Firestore version does not implement the *Durable* view; the average delay for the *Durable* view in the Hercules app was 2.2ms.

For the Hercules app, the *Authoritative* delay is roughly the RTT between the client and server, and the *Visible* delay is around two times the RTT, as expected. The Cloud Firestore app's *Authoritative* delay is 2x the Hercules app's delay, although this is not quite an apples-to-apples comparison—the Cloud Firestore backend may be replicating updates or performing other useful work with the added time. Its *Visible* delay is 9.5x the Hercules app's delay, with a large variance, due to the batching described in Section 3.6 that is required to avoid dropped writes.

3.7.3 Effect of Client Rebase Batching

Client rebase batching allows Hercules deployments to maintain good performance as system load increases. Figure 3.14 shows how rebase batching prevents performance from collaps-

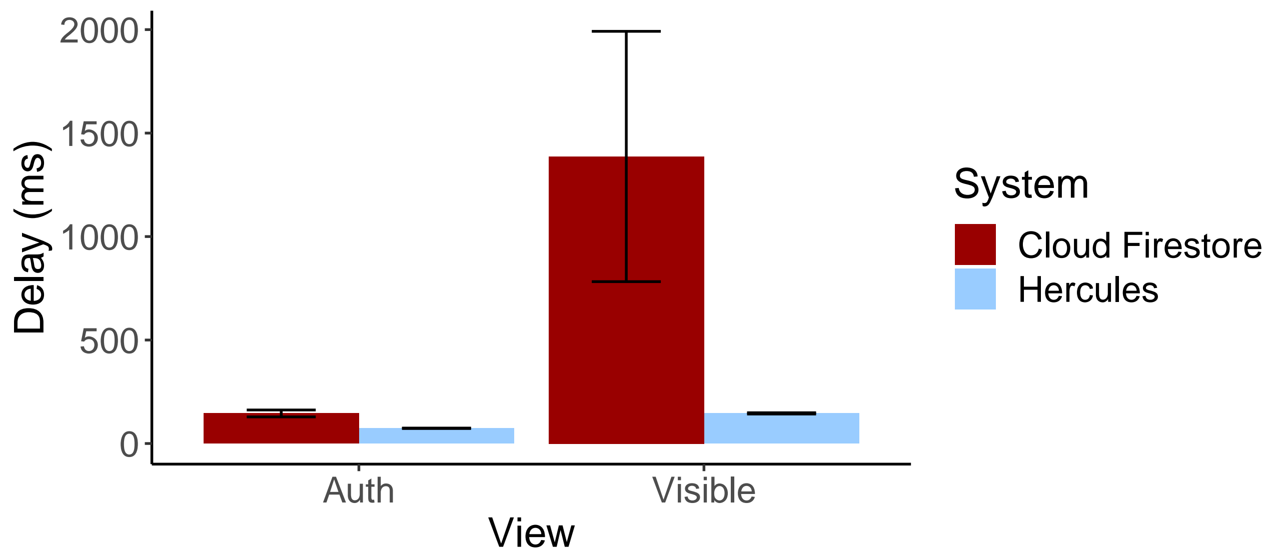

Figure 3.13: The average delay before a submitted operation in the spreadsheet app is reflected in the client's state views for both the Hercules and Cloud Firestore versions.

ing as a Hercules deployment expands to include more clients. The graph plots the time to complete an experiment in which each client submits operations and then waits for all operations from all clients to be reflected in its *Visible* view. For this experiment, we used our benchmark application, with a 100KB shared byte array; each client submitted 1000 operations, sleeping for 20ms between each operation, and each operation performed 500 array increments. We ran each configuration five times, plotting a point for each run and lines showing the mean.

Without rebase batching, every remote operation triggers a rebase. As load increases, rebase times exceed the rate of incoming operations, and clients become trapped in a vicious cycle of continually rebasing. This vicious cycle results in much higher experiment completion times, which occurred occasionally with eight clients and consistently with ten clients. With rebase batching enabled, clients perform rebases at a lower, fixed frequency, instead of every time a remote operation arrives. As a result, they remain responsive as load increases, and experiment completion times stay low.

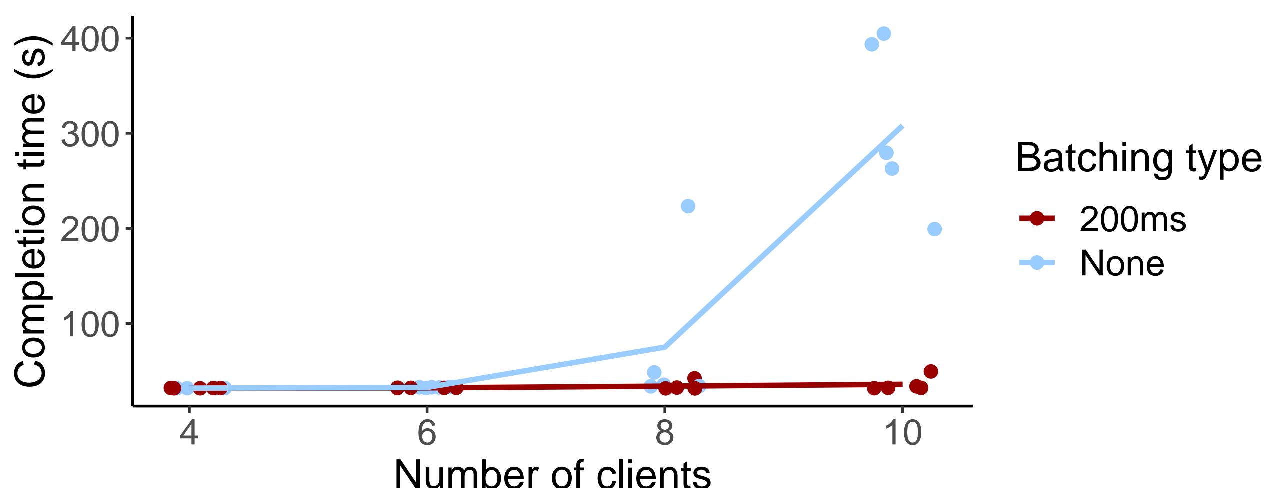

Figure 3.14: **Effect of batching as the number of clients increases.** Rebase batching allows clients to cope with the added load introduced by more clients.

3.7.4 Recovery from Client Failures

Figure 3.15 shows how one client's failure impacts another client's state views. The graph plots the delay between when a client submits an operation and when it sees that operation reflected in its state views. For this experiment, we ran the benchmark application with four clients. The shared byte array was 10KB in size, and each operation performed 1000 array increments; each client slept for 20ms between operation invocations.

At around the 20-second mark, another client is killed. The measured client's *Visible* view experiences a delay spike at that point, as the system waits for the visibility set timeout to kick in, but the *Authoritative* view is unaffected. Note that even though there is a longer delay before updates make it into the *Visible* view, the client can still read that view with low latency.

3.8 Related Work

Global Sequence Protocol (GSP) [14] is a model for replicated shared data that represents shared state as a sequence of updates. Clients distinguish between a *known* prefix of globally-

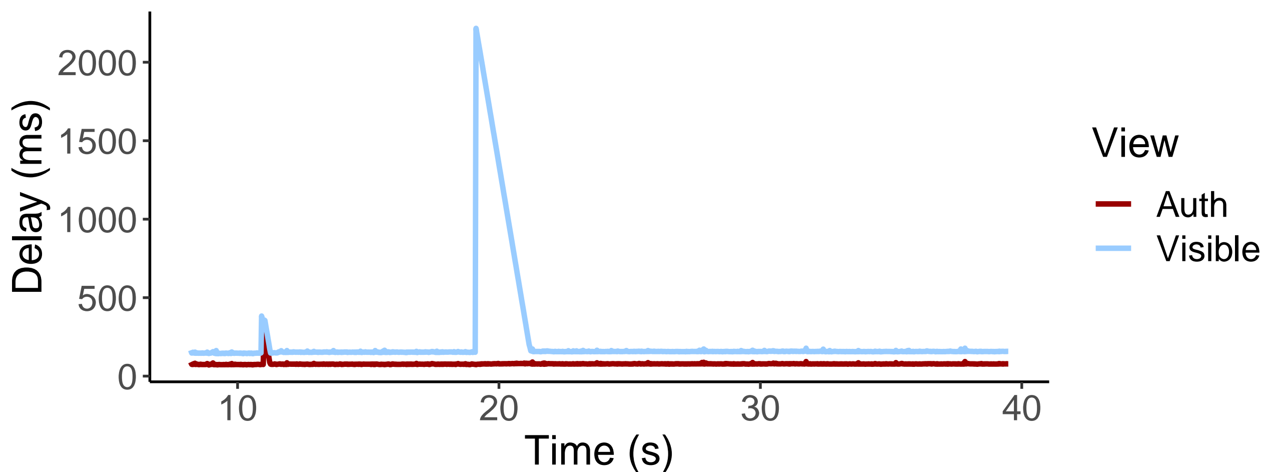

Figure 3.15: **Effect of client failure on view staleness.** When one client fails, the measured client's *Authoritative* view is unaffected, but its operations are delayed from being applied to its *Visible* view until the visibility set timeout kicks in.

agreed-upon updates and a *pending* sequence of local updates. Hercules's model of shared state resembles GSP's, and Hercules makes use of GSP's insight that "including application-specific update operations in the data model is a powerful trick." In contrast to GSP, Hercules exposes additional sub-sequences of updates, one of which (the *Visible* view) requires substantial backend support, and Hercules emphasizes providing its state views on-demand with low latency.

The Correctables abstraction [50] introduces the concept of executing operations on shared state with multiple consistency levels and incrementally providing feedback to applications. It provides that feedback in the form of per-operation callbacks, which we argue is the wrong abstraction. With per-operation feedback alone, apps cannot determine which portions of shared state reflect uncertainty. Also, if apps need immediate access to strongly consistent snapshots of state when rendering output, then they need to manually schedule strongly consistent read operations in the background and cache the results.

Bayou [100] introduces many ideas that Hercules builds upon. It provides two views of its data store reflecting different consistency levels, and its sample applications use those

views to expose uncertainty to users. The differences between Hercules and Bayou result from changes in the mobile computing landscape over the last several years. Bayou assumes that network partitions are a common occurrence, and its concerns with latency are limited to ensuring that clients can make progress during partitions. In contrast, Hercules targets an environment in which full connectivity is the normal case, and client read latency is of paramount importance. These assumptions motivate Hercules's *Visible* and *Submitted* views, respectively.

Multiple systems build on Bayou to provide flexible consistency guarantees [108, 10], but these systems do not ensure low client read latencies, and they do not provide multiple simultaneous views of the same data item. Other systems that provide multiple consistency levels [68, 70] similarly expose only one view at a time.

Some systems use speculation techniques to mask network latency and improve the responsiveness of distributed apps in particular domains. VNC/SRD [63] adds client-side speculation to a VNC remote desktop system, and Outatime [66] applies server-side speculation and client-side error correction to a cloud gaming system. Instead of exposing uncertainty to users, these systems directly display a tentative view, although Outatime includes multiple techniques for compensating for mispredictions in the client.

Several storage systems target distributed mobile applications. Diamond [109] is one such system; Section 3.1 discusses how Diamond's interface makes exposing uncertainty difficult, and how Hercules addresses this problem. Another system in this space is Simba [40]. Like Diamond, Simba supports multiple consistency levels but only provides a single view of any particular data item to application clients, preventing them from exposing uncertainty to users. Diamond and Simba both also require a client's own operations to go through durable storage before being reflected in its reads, which risks violating the responsiveness requirements of real-time interactive apps, and they do not provide any information about the visibility of one client's operations to other clients. Other work targets distributed file systems [76, 101, 110], where accesses are more sporadic and coarse-grained, and the focus is on conserving bandwidth.

Many areas of active research in distributed systems are complementary to Hercules. Geo-distributed storage systems with strong consistency guarantees [9, 20, 59] could be used as the backend component of a Hercules deployment, and even systems that sacrifice consistency for performance [22] could be used if they provide stale, strongly consistent snapshots. On the client side, techniques from conflict-free replicated data types [93] could minimize the computing burden by avoiding the need to re-compute state views.

3.9 Conclusion

Real-time interactive apps need to respond instantly to user input, and to do so, they must expose uncertainty in shared state. Existing distributed storage systems only provide a single view of shared state, making it difficult to identify and expose uncertainty; apps must manually track uncommitted operations, maintain local copies of shared state, and synchronize that local data with the storage system's view when it changes. Hercules simplifies real-time interactive app development by providing a multi-view cache of shared state. Apps need only compare Hercules's views to identify and expose uncertainty. Its views trade off consistency and staleness, so apps can always query them with low latency, and they span the extreme ends of the consistency/staleness spectrum, letting apps meet the instant response needs of modern devices. Hercules provides a configurable rebase batching optimization that lets developers control the tradeoff between client load and stale views' update frequency, and its failure-handling protocol allows it to track operations' visibility without being blocked by crashed or disconnected clients. Our app case studies showed that Hercules makes it easy for developers to expose uncertainty when building a variety of different real-time interactive apps; our evaluation showed that Hercules's rebase batching preserves client performance as deployments scale to include more clients, and that its failure-handling protocol ensures the liveness of a client's views as other clients disconnect.

Chapter 4
MARVIN

This chapter moves away from shared data management to address a problem that every mobile app encounters, regardless of whether its data is shared or static: the problem of how to work with large amounts of data in the face of limited device memory. Existing mobile platforms give each app a fixed maximum memory budget and kill apps when memory runs out, putting the burden on app developers to manually move data between memory and disk and optimize their apps' memory footprints. This chapter presents *Marvin*, an Android memory manager co-designed with Android's Java runtime that re-introduces swapping to mobile platforms. Whereas Diamond and Hercules introduced new interfaces for developers to use, Marvin's swapping mechanism is completely transparent: developers use the existing Java interface for allocating objects, and Marvin swaps objects in and out of memory as needed. This chapter is based on work published as a University of Washington Technical Report [65].

4.1 Introduction

Over the past decade, mobile apps have become bigger and more complex [78], far outpacing increases in mobile device memory [12]. This trend has increased memory pressure on mobile operating systems as apps compete for limited space. Going forward, mobile OSes must more efficiently share memory across demanding apps, or user experience will suffer.

Unfortunately, while mobile apps have become more sophisticated, mobile memory management remains in its infancy. Today's popular mobile OSes set a fixed upper bound on memory for each running application (e.g., 1.4GB for iOS running on an iPhone X [97] and 512MB for Android running on a Google Pixel XL). They never overcommit memory;

instead, they kill running applications and restart them later.

This simplistic approach worked well when mobile apps were small and largely stateless. However, it is unsustainable as mobile OSes replace desktop ones (e.g., Android is now the most used OS in the world) and mobile apps replace desktop counterparts (e.g., Google Docs and Word Online replacing Microsoft Word). Today's apps already do not fit into their memory allocation, so they manually swap objects between memory and local storage or use libraries to meet their needs [26]. Because apps are increasingly likely to be killed due to memory pressure, they must also continuously save execution state to disk and strive to minimize their start-up times to cope with frequent restarts. Despite significant engineering effort [90, 77, 57], it still takes several seconds to kill and restart popular apps.

Improving mobile memory management is difficult. Mobile apps run in high-level language runtimes (e.g., Swift, ART), which limit OS insight (e.g., working set estimation is impossible) and are notoriously difficult for OS-level memory managers to work with [52]. Further, mobile apps often allocate large amounts of memory quickly (e.g., when starting, or for cloud downloads); unless the OS keeps a large pool of free memory, this is easier to accommodate by killing entire applications. Finally, touch-based interfaces impose strict latency requirements, which swapping to disk cannot meet.

In this chapter, we improve mobile memory management with a key observation: unlike other operating systems, mobile OSes run all of their apps in a *common* language runtime. For example, all apps running on Android must run in the Android Runtime (ART). This difference lets us co-design the language runtime to assist the mobile OS in optimizing memory management instead of hindering it. Due to its knowledge of memory usage, the language runtime becomes an ideal place for mechanisms that can better manage memory.

This chapter demonstrates the value of leveraging the runtime for OS tasks. We present *Marvin*, a new memory manager for Android that efficiently supports memory overcommit while meeting the strict performance requirements of mobile apps. Marvin implements most memory management in the language runtime, which has more insight into an application's memory usage. Marvin relies on the operating system only for cross-application resource

allocation.

By integrating with the language runtime, Marvin can offer three new features that enhance memory management:

- A new swap mechanism, which performs *ahead-of-time* swapping to disk to meet app latency requirements.

- A new object-level working set estimator, which separates garbage collector (GC) and app accesses, and avoids false sharing with object-level access tracking.

- A new bookmarking garbage collector [52], which tracks exact liveness data without accessing swapped-out objects.

We implement a prototype of Marvin by modifying the interpreter and compiler of the Android Runtime (ART). Experiments show that our Marvin prototype is able to run more than 2x as many concurrent apps as Android, and that Marvin can reclaim memory over 60x faster than Android with a Linux swap file can allocate memory under memory pressure.

4.2 Limitations of Modern Mobile OS Memory Resource Management

Although mobile OSes may be based on traditional OSes (e.g., Android and Linux), they diverge in two important ways: (1) for each app, they bound memory usage to a fraction of physical memory (e.g., 512MB on a 4GB device), rather than letting apps allocate as much memory as they need, and (2) they kill applications when physical memory runs out rather than overcommitting memory through paging or other mechanisms. To motivate our work, we ran experiments with popular apps that show the reasoning and cost for these design decisions. We ran all experiments on a Pixel XL phone with 4GB RAM and a quad-core Qualcomm Snapdragon 821 CPU.

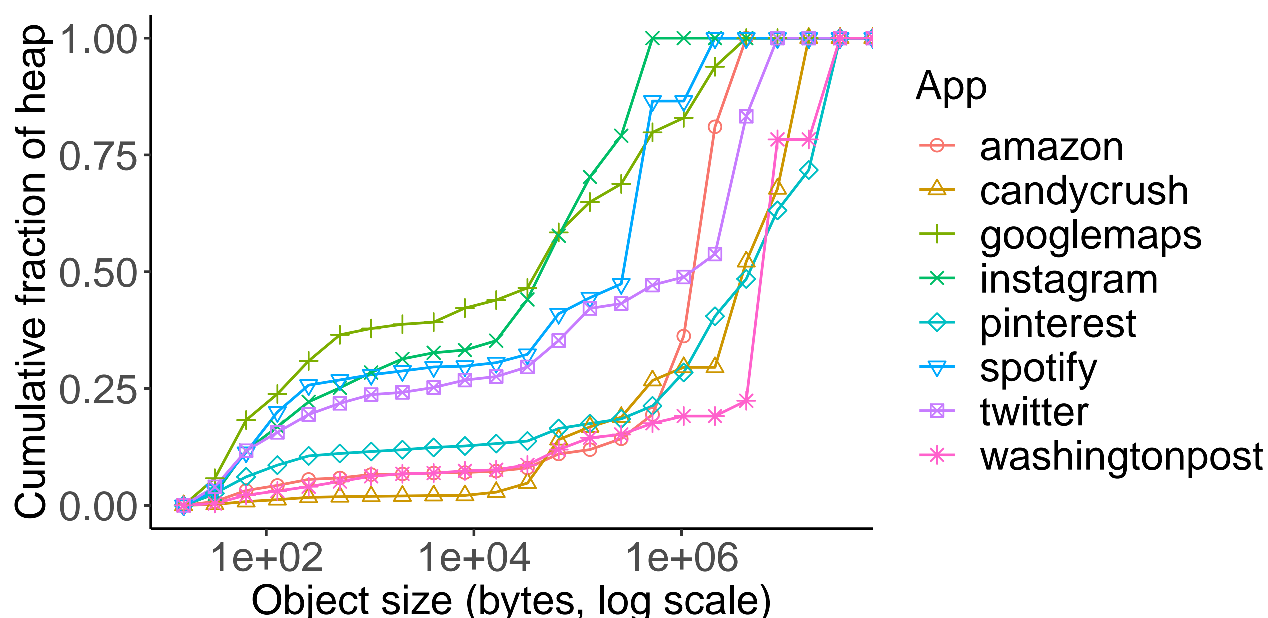

Figure 4.1: CDF of object size and heap percentage occupied by objects that size or smaller. Popular Android apps have a bimodal distribution where most objects are either significantly smaller or larger than a 4KB page.

4.2.1 Fixed Memory Allocation

Mobile OSes have poor insight into app memory usage. The runtime garbage collector regularly touches all objects and moves objects for heap compaction, and the OS cannot distinguish this activity from app accesses. Mobile apps also access language-level objects, which vary in size, while the OS can only track memory accesses at page granularity. To understand the impact of object-level accesses on page-sized access tracking, we measured the size of objects in popular apps. Figure 4.1 shows a CDF of the size distribution. Most objects are not page-sized (e.g., for some apps, up to 40% of objects are less than 4KB in size), so the OS cannot accurately track their usage.

Without good insight into app memory usage, today's mobile OSes allocate all apps a fixed memory budget. On Android, this memory limit is the same whether an app is in the foreground or background. Android attempts to minimize the memory footprint of apps using

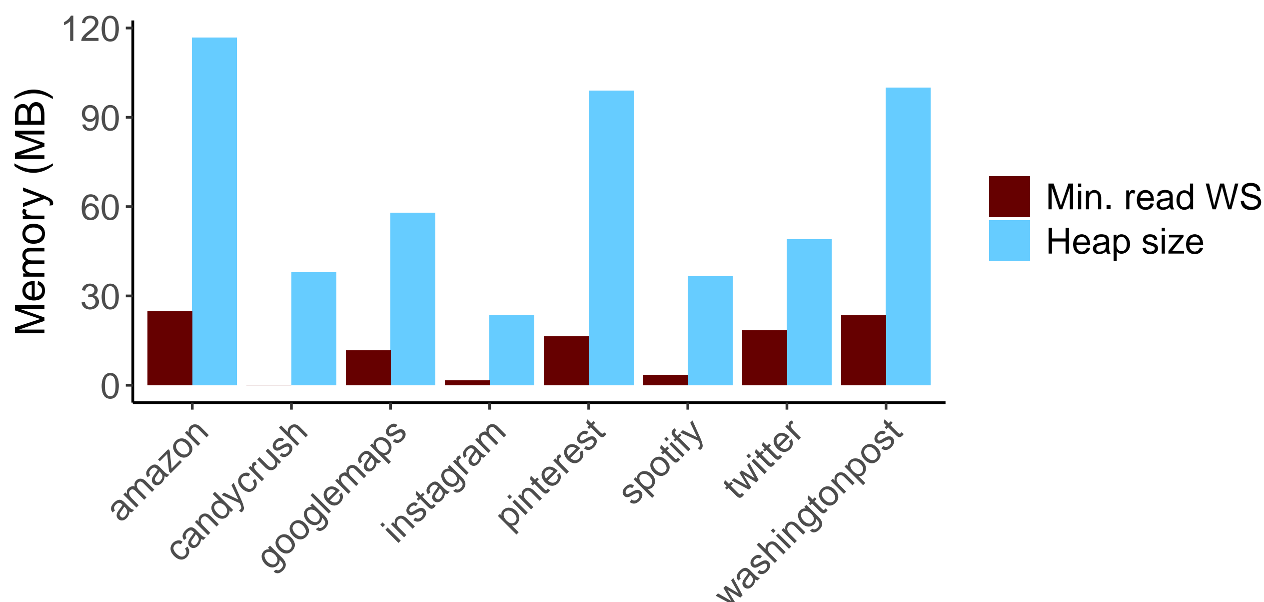

Figure 4.2: **The cost of fixed allocation.** Each bar shows the total Java heap size of a popular app alongside its minimum Java working set during active use. While apps have large memory footprints, they do not use most of that memory, which could be better utilized for running another app.

techniques such as forking all apps from a single "zygote" process with copy-on-write pages. These techniques reduce duplication of framework data structures and shared libraries but do not impact application objects in the Java heap. Using Marvin's object-level working set estimator, we measured the working set of popular apps. Figure 4.2 shows that although the heap footprint of these apps is large, their working sets actually account for a small fraction of their total heap size. This rarely accessed memory would be better utilized keeping other apps alive, rather than wasting space not being used.

Popular apps often have large memory footprints but small working set sizes because they cache as much as possible from the cloud. This caching improves performance at no cost to the app, but it leads to poor memory utilization, and choosing the correct cache parameters is difficult [81]. Worse, modern applications frequently exceed their memory budgets. Coping with this problem requires applications to implement manual swap-to-

storage, which adds significant programming complexity to them, as demonstrated by the amount of documentation and tutorials on this topic [25, 26]. While caching libraries like Glide [13] and Fresco [95] are helpful, they do not apply to all memory objects. Therefore, today's apps use a complex combination of libraries and manually shuffling data between memory and disk.

4.2.2 No Memory Overcommit

Today's mobile OSes kill applications rather than swapping to disk when physical memory runs out. They take this approach because mobile apps must respond to user input within hundreds of milliseconds, so traditional swap mechanisms impose too much latency. To measure the effect of swap on memory allocation, we enabled a Linux swap file on our Android test device [96] and measured the amount of time required to allocate 512MB when the Android OS had free memory and when it did not. Figure 4.3 shows the progress of the memory allocation over time. With memory available, the OS allocated all 512MB in 450ms; however, under memory pressure, it took almost 8 seconds for the OS to allocate the same amount. Such high allocation latency would be unacceptable if an app were allocating memory in response to user input.

Unfortunately, killing and restarting apps comes at a cost. As shown in Figure 4.2, modern apps have large memory footprints, and a restarted app must fetch all of its cached data from the network or disk. We measured the amount of time needed to restart popular apps and compared it to that needed to fetch their entire checkpointed memory image from disk. As shown in Figure 4.4, restarting apps takes 4-27x longer than fetching the all of the app's memory from disk.

The ability to kill and restart apps at any time also imposes a programming burden on app developers. Modern OSes give apps a limited time budget to perform cleanup before being killed. This limit leads apps to constantly write state to storage; in fact, Android encourages it [24]. Such constant checkpointing in response to app lifecycle events adds programming complexity, a challenge described in prior work [34]. Not only do app developers have to

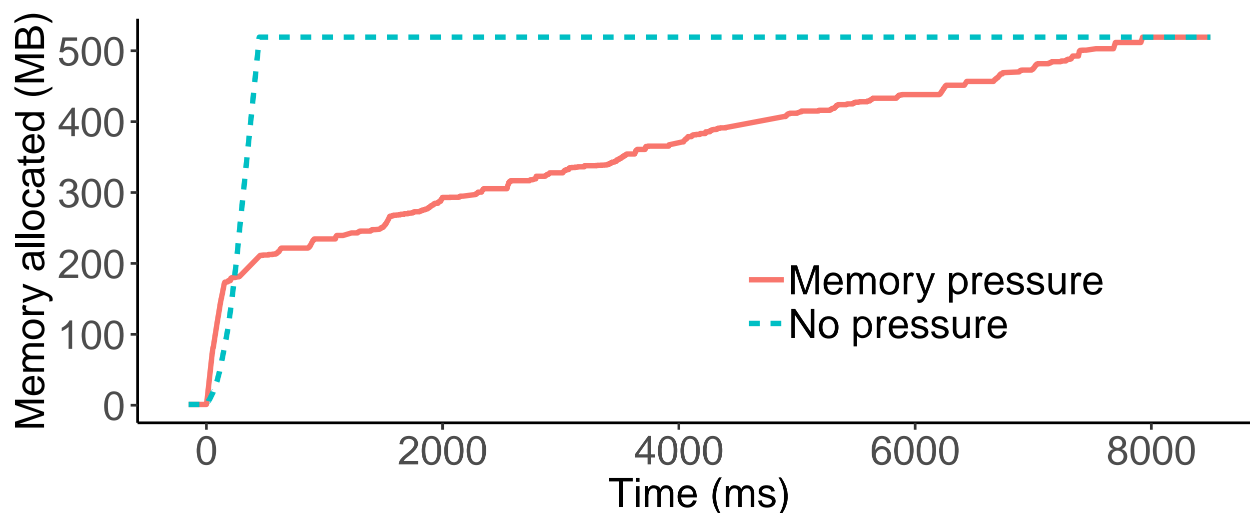

Figure 4.3: **Progress over time of a memory allocation on Android with and without swap.** Android allocates all 512MB of memory in 450 ms when memory is free, while swap increases that time to almost 8 seconds.

manage the checkpointing process, they have to correctly use a variety of mechanisms to do so with good performance [27]. The programming effort required to prepare for unexpected app deaths is an additional cost that app developers must pay.

4.3 Our Approach

The primary barrier to improving memory resource management in mobile operating systems is the OS's lack of insight into the language runtime. To overcome this barrier, we *co-designed* the language runtime and the mobile OS. Mobile operating systems are uniquely suited to such co-design because, unlike their desktop counterparts (e.g., Linux), they force all applications to use the same language runtime.

Marvin's design focuses on Android (and the Android Runtime (ART)), which is now the most popular OS in the world [71]. Using the language runtime, Marvin manages memory entirely at object granularity, tracking, swapping, reclaiming and faulting in entire objects. This section describes in more detail the barriers to better memory resource management in a mobile OS and how Marvin addresses those challenges.

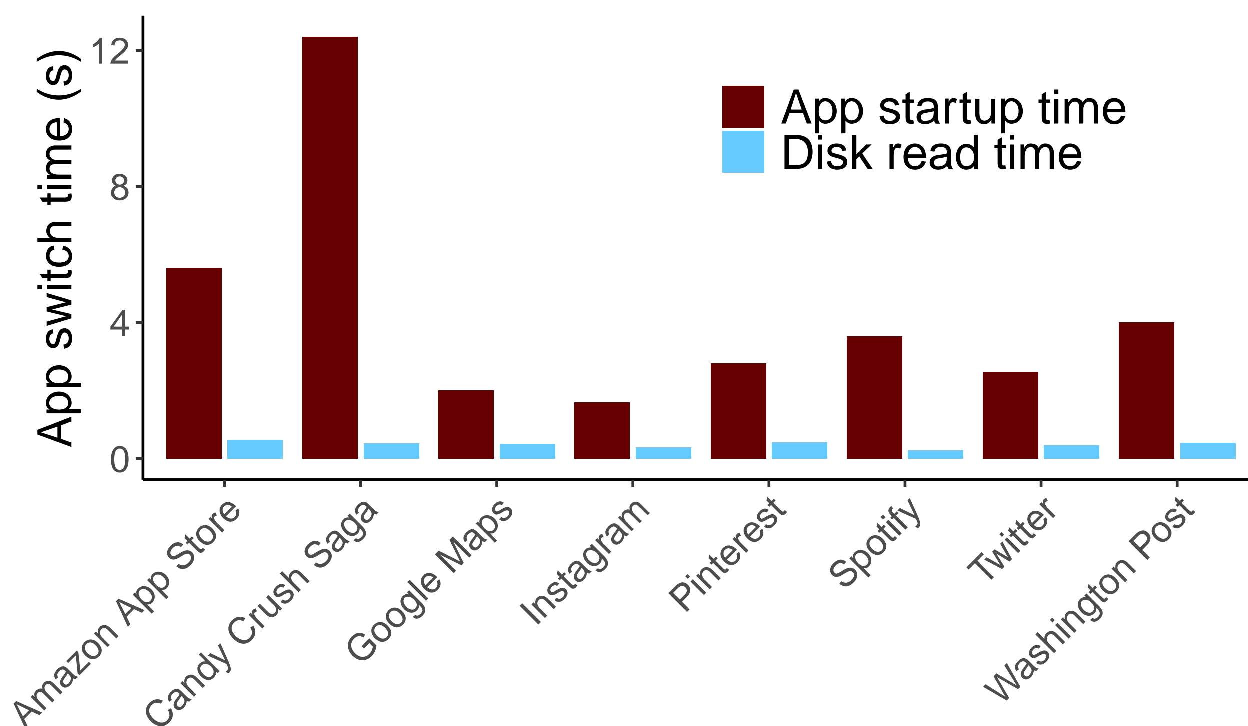

Figure 4.4: **The cost of re-starting apps.** Modern mobile OSes kill apps when memory runs out rather than swapping to disk. This wastes significant time for popular apps, which take anywhere from 4x to 27x longer to restart than to read *the entire app memory image from disk.*

4.3.1 Object-Level Working Set Estimation

The first step to better memory resource management is a better understanding of each application's working set. Thus, Marvin implements language-aware working set estimation in the language runtime, which tracks app reads and writes at object granularity. Marvin uses this mechanism to identify candidates for ahead-of-time swap (Section 4.3.2) and separate garbage collector accesses from app accesses (Section 4.3.3). Lacking hardware access bits to help with this tracking, Marvin implements software access tracking in both the ART interpreter and compiler, as modern mobile language runtimes run both interpreted and compiled code.

4.3.2 Ahead-of-time Swap

As noted, swapping to disk when the OS needs memory is not feasible for mobile OSes and their touch-based apps. Marvin takes a different approach. While traditional swapping mechanisms write to disk when memory is needed, Marvin uses a new *ahead-of-time* swap technique. This technique saves memory to disk *before* it is needed and then reclaims those pages under memory pressure. Ahead-of-time swap separates swapping to disk from reclaiming memory; thus, we distinguish between *saved* objects, which have been copied to disk but still reside in memory, and *reclaimed* objects, which no longer reside in memory but are only on disk.

Swapping objects before they are needed leaves a large pool of clean memory that the OS can quickly reclaim and reallocate. While this technique lets apps continue using memory until the OS reclaims it, whenever the app dirties a page, the swap mechanism must update the on-disk copy before the OS can reclaim it. Due to this trade-off, Marvin prioritizes swapping objects that are infrequently or never written.

4.3.3 Bookmarking Garbage Collector

Like traditional swapping, ahead-of-time swapping is affected by friction with the language-level garbage collector. As noted by Hertz et al. [52], the garbage collector can inadvertently page in memory when walking the object heap to look for unused objects. With ahead-of-time swapping, the garbage collector can also inadvertently dirty pages when updating references, causing unnecessary writes to disk. Marvin solves this problem by integrating a modified bookmarking garbage collector [52] into the Android Runtime.

Marvin's swap mechanism leaves *stubs* – analogous to bookmarks – for each reclaimed object that detail the objects' references to other objects. Using these stubs, its garbage collector can process a reclaimed object during a mark-and-sweep run without faulting in the entire swapped object. Marvin's swap mechanism can further optimize swapping from disk by dropping dead objects without faulting them in.

4.4 Marvin Overview

Marvin is a new mobile memory manager that supports flexible memory allocation between apps and memory overcommit through swapping. Marvin includes components in the language runtime and OS, which are co-designed to provide better memory management. This section gives an overview of both.

4.4.1 Design Goals

Marvin's design meets the following goals:

1. **Fast memory allocation.** Marvin must allocate memory quickly on-demand, avoiding disk accesses on the critical path for memory allocation.

2. **High memory utilization.** Marvin must provide the illusion of unlimited memory, provided working sets do not exceed the size of physical memory.

3. **Minimal overhead.** Marvin must impose low runtime overhead and require no app code changes.

While the last two goals are common to all memory management systems, existing mobile platforms sacrifice high memory utilization for fast memory allocation. Marvin aims to achieve all of these goals.

4.4.2 Marvin System Model

Marvin assumes a systems environment that meets three requirements: (1) all apps are written in a single managed language (e.g., Java), (2) all apps run in a single managed language runtime (e.g., ART), and (3) the runtime performs garbage collection or some form of automatic memory management. Marvin's design targets Android, which meets all of these requirements. Android also runs some libraries using native code; however, Marvin is

not needed to manage their memory because they do not have the same issues with OS-level memory management as managed languages.

Marvin's optimizations could apply to other operating systems as well (e.g., iOS). For example, Swift uses automatic reference counting as an alternative to garbage collection, so it would require a bookmarking reference counter that can track references without faulting in the entire object.

Marvin runs unmodified Android apps on ARM64-based devices. Android distributes apps in a bytecode format called DEX. ART runs DEX bytecode directly in an interpreter and also compiles DEX to native ARM64 instructions both at install time (ahead-of-time, or AOT, compilation) and at runtime (just-in-time, or JIT, compilation). Marvin modifies both the interpreter and compiler.

4.4.3 Marvin Architecture

Marvin has two key components: (1) the *Marvin Kernel* (MK), a modified Android/Linux kernel, and (2) the *Marvin Runtime* (MRT), a modified ART. Most memory management occurs in MRT; it performs working set estimation, ahead-of-time swapping, and bookmarking garbage collection. MK's sole responsibility is to balance memory allocation among apps by deciding when and from which app to reclaim memory.

Marvin performs working set estimation and swapping at object granularity; however, there is no CPU support for object-level access bits and memory faults. As a result, Marvin implements software object access tracking and faulting in the MRT interpreter and compiler. The interpreter marks access bits and checks for swapped-out objects as it runs DEX bytecode; the compiler inserts that functionality as additional ARM64 instructions. Marvin reserves four bytes in each object header and uses those bytes to store swapping metadata and access bits. Implementing these features in software imposes an app overhead, which we quantify in Section 4.8. Using hardware features in future mobile devices could significantly reduce this overhead.

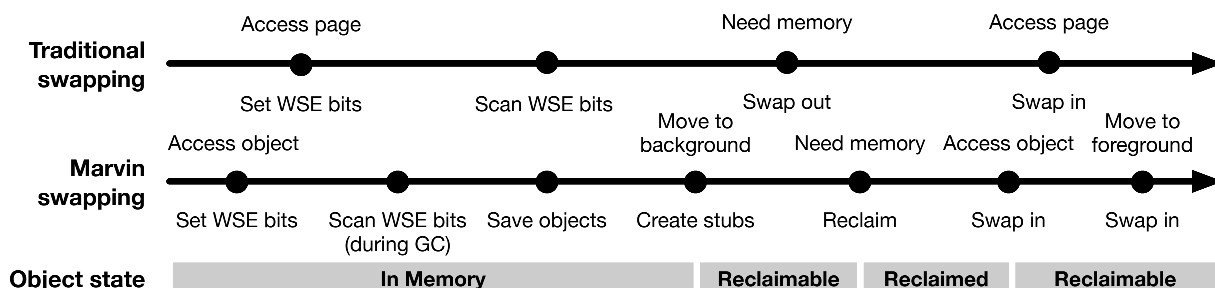

Figure 4.5: A timeline of actions performed by Marvin's swap mechanism as compared to traditional (e.g., Linux) swap mechanisms. Events are listed above the timeline while Marvin's actions in response are listed below.

4.4.4 Marvin Memory Management Timeline

Objects managed by Marvin move through several states over time, driven by app behavior and app lifecycle events. Figure 4.5 illustrates these events and states and compares Marvin's swapping to a traditional swap mechanism. When an app first starts, MRT begins tracking its working set. It identifies objects that are suited for swapping by examining whether they are cold (have not been read or written recently by the app). MRT begins saving checkpoints of those objects to disk in the background. We refer to an object with a saved checkpoint as a *saved* object.

When the app moves from foreground to background, MRT pauses app threads and creates stubs, small proxy objects that add a layer of indirection over swap candidate objects. Stubs ensure that Marvin can intercept accesses to objects and fault them back in, if necessary. Once MRT creates a stub for an object, that object becomes *reclaimable*; the object is still memory-resident, but MK can reclaim its memory at any time. When MK reclaims an object, it enters the *reclaimed* state; only the object's stub remains in memory, and the object's checkpoint will need to be faulted back into memory before the object can be accessed again. The garbage collector uses only the stub and need not fault the object back into memory.

4.5 Marvin Core Mechanisms

As noted in Section 4.3, Marvin's key features are ahead-of-time swap, language-aware working set estimation, and bookmarking garbage collection. Designing these features required addressing three challenges: adding a layer of indirection for object references, coordinating between the OS and runtime, and interposing on object accesses. This section describes Marvin's mechanisms for addressing these challenges.

4.5.1 Stubs for Object Reference Indirection

According to the Java language specification, object references are opaque. However, in practice, object references in the Android Runtime are direct pointers to the heap memory holding the referenced object. This design requires Marvin to inject a layer of indirection to implement features like software object faulting and the bookmarking garbage collector. Marvin creates this layer of indirection using special objects, called *stubs*. Each stub contains a pointer to its underlying object along with a copy of each reference held by the object. All references to an object point instead to its stub, and only the stub holds a pointer to its underlying object. Accessing an object through a stub adds overhead, so Marvin creates stubs only for objects that are cold and at least 2KB in size.

When creating a stub for an object, Marvin moves the object to a separate page-aligned region of memory and then redirects all references to the object to point to its stub instead. These tasks require that all app threads be paused. Therefore, they can be performed more efficiently if Marvin can create stubs for many objects at once, using a single scan of the heap to redirect all affected references. As a result, Marvin periodically executes a heap task that pauses all app threads and creates stubs, and it executes this heap task only when the app is in the background and its threads can be safely paused. Stubs are only created once for each object, so the cost is low, especially because Marvin does not restart apps frequently.

4.5.2 Reclamation Table for OS-Runtime Coordination

Modern mobile platforms have multicore processors that let system services run concurrently with apps. In this environment, the OS should be able to reclaim memory quickly from a running app without scheduling the app's threads for execution. However, the OS cannot simply seize memory from an app whose threads are not scheduled—a pointer to the memory in question may be present in an app thread's stack or registers, waiting to be used as soon as the thread is scheduled once again. As a result, the OS and runtime need a way to coordinate concurrent accesses to objects so the OS does not try to reclaim one that the runtime is accessing.

Marvin uses a shared-memory *reclamation table* to provide this coordination. MRT populates the table with reclaimable objects, and MK uses it to identify memory to reclaim. Each reclamation table entry is a small, fixed-size data structure that holds the address of an object, its size, a set of flags indicating whether the object is memory-resident and the entry is valid, and a set of bits used for locking by the runtime and OS. To reclaim an object, MK first acquires an exclusive lock on the object's reclamation table entry. Similarly, whenever an app thread prepares to access an object, MRT acquires a shared lock on the reclamation table entry.

4.5.3 Object Access Interposition

All of Marvin's features require the runtime to interpose and perform specific tasks whenever an app accesses an object. On every object access, MRT must set read and write bits for working set estimation; check for the presence of a stub and redirect the object access through the stub if necessary; and fault in the object if it has been reclaimed. It must also set a dirty bit whenever an object is modified so the ahead-of-time swap mechanism knows which objects need to be saved, and it must update stubs whenever reference member variables in their corresponding objects change to support the bookmarking garbage collector.

Android apps execute both as DEX bytecode running in an interpreter and as compiled

native code running directly on the hardware, and Marvin must interpose on all object accesses in both kinds of code. As a result, Marvin features a set of paired interpreter and compiler modifications that add the required object access interposition. For each additional task performed by the interpreter when it accesses an object, there is a corresponding change to the compiler, adding assembly instructions performing the same task to compiled code.

4.6 Marvin Memory Management

This section describes how we use Marvin's mechanisms (stubs, the reclamation table, and object access interposition) to design the features that make up Marvin's memory management system.

4.6.1 Working Set Estimation

MRT performs object-granularity working set estimation by maintaining two access bits in each object header, a read bit and a write bit, and scanning those access bits.

Setting access bits. MRT uses object access interposition to set an object's read and write bits whenever that object is read or written from either interpreted or compiled code. It avoids including garbage collector reads in its working set estimation by setting a flag in the object header when the garbage collector is visiting an object and leaving the read bit untouched if that flag is set.

Access tracking in MRT is performed on a best-effort basis to minimize its overhead: MRT uses non-atomic operations with relaxed memory ordering semantics when setting read and write bits. As a result, concurrent reads and writes to the same object could result in a lost update to one of the access bits. An update to the read bit could also be lost if an app thread reads an object that the garbage collector is processing. These optimizations may decrease swapping performance if the estimated and actual working sets differ significantly, but they do not affect correctness.

Scanning access bits. MRT periodically walks the heap and uses the Clock algorithm [23] to track each object's long-term usage. Each object header holds two four-bit *shift registers*, one for reads and the other for writes. The time between heap walks constitutes an access-tracking "round," and each shift register tracks whether the object was read or written in the last four rounds. During a heap walk, MRT updates an object's shift registers and then clears the object's access bits.

MRT piggybacks off of garbage collection to scan access bits, since GC requires walking the heap anyways. Some of ART's GCs only walk subsets of the heap, so MRT limits its access bit scanning to full-heap collections. It also periodically invokes GC to ensure up-to-date working set estimates in the absence of app activity.

Producing the working set. As MRT walks the heap and scans access bits, it tabulates the app's working set. Our current MRT implementation considers an object part of the working set if it has been written within the last four access-tracking rounds. The precise policy is an implementation detail that can be easily changed.

4.6.2 Ahead-of-Time Swapping

In Marvin's ahead-of-time swap mechanism, MK reclaims objects and decides which apps to target for reclamation. MRT performs all other functions, including saving object checkpoints to disk, restoring reclaimed objects, and preventing the operating system from reclaiming objects in use by app code.

Saving objects to disk. MRT identifies suitable objects for swapping (i.e., cold objects) using its working set estimation feature, and it proactively saves checkpoints of them to a swap file on disk so they can be reclaimed quickly under memory pressure. MRT saves objects to disk in a periodic heap task that runs on a background thread concurrently with app code.

After app code modifies an object, MRT must save an updated copy of that object to disk.

It does not need to save the updated copy immediately as long as it prevents the kernel from reclaiming the object while it is "dirty." To do so, MRT maintains a *dirty bit* in the object header. It uses object access interposition to set this dirty bit whenever app code writes to an object, and its object-saving heap task clears this dirty bit when saving the object to disk. MK checks dirty bits when looking for objects to reclaim and avoids reclaiming dirty objects. MRT and MK use strong memory-ordering semantics when reading and writing the dirty bit to ensure that no modifications to objects are lost.

MRT begins saving swap candidate objects to disk even before those objects have had stubs created for them. Once MRT creates a batch of stubs, those objects become immediately reclaimable without requiring further disk I/O.

Reclaiming objects. MK selects apps to target for reclamation and reclaims objects from the MRT instances corresponding to those apps. It never targets the foreground app, in order to minimize user-visible stuttering. After selecting an MRT instance to target, MK scans the MRT instance's reclamation table until it finds an entry for an object that is neither dirty nor locked by the runtime. MK then locks that entry and reclaims the object's pages. It continues scanning the reclamation table and reclaiming objects until it has harvested the desired amount of memory from the MRT instance.

Each MRT instance ensures that MK does not reclaim an object currently being accessed by its app code. To do so, it uses object access interposition to detect whenever a reclaimable object is being read or written, and it locks the object's reclamation table entry before the access and unlocks its entry after the access.

Restoring objects. MRT restores reclaimed objects either eagerly or on-demand. Either way, whenever MRT restores an object, it locks the object's reclamation table entry, copies the saved checkpoint data of the object into memory, and copies any modified references from the object's stub into the object itself. This last step is necessary because references in the stub may have been modified by the garbage collector while the object was not memory-

resident.

Marvin's eager object restoration uses app lifecycle information to restore objects before app code needs them. We implemented a simple eager restoration policy, where an MRT instance restores all reclaimed objects when it transitions to the foreground. This policy ensures that no user-perceptible delays or stuttering result from swapping activity. Our design is flexible and could support more advanced policies; for instance, the runtime could predict which objects are likely to be touched immediately after a foreground transition and restore those objects first, trading off a shorter pause time in exchange for the risk of user-perceptible stuttering.

If an object has not been eagerly restored, MRT restores it on-demand when app code accesses it, a process that we call *software object faulting*. Whenever app code accesses a reclaimable object, MRT uses object access interposition to check if the object is memory-resident by inspecting a bit in its reclamation table entry. If not, MRT executes an object fault handler that performs a procedure call into its C++ object restoration function.

4.6.3 Bookmarking Garbage Collector

A tracing garbage collector touches every object in the heap (or a subset of the heap), causing live objects to be swapped back into memory even if app code is not using them. Marvin's garbage collector avoids touching reclaimed objects by storing an object's references inside its stub and using the stub during the mark phase of garbage collection. Stubs play a similar role as bookmarks in the bookmarking garbage collector [52].

During the mark phase, the garbage collector maintains a *mark stack* and repeatedly pops an object from the mark stack, marks all its references, and pushes those references onto the mark stack. Marvin's garbage collector checks whether an object is a stub when it pops the object from the mark stack; if so, it reads the references off the stub instead of accessing the underlying object.

For the garbage collector to use stubs in place of their objects, MRT must ensure that the stub of a memory-resident object has up-to-date copies of the object's references. It uses

object access interposition to update the stub of a reclaimable object whenever Java code modifies one of the object's reference member variables.

MRT must also properly clean up after any saved objects that are freed by the garbage collector. MRT records when saved objects have been freed by the garbage collector, and when the fraction of the swap file consisting of freed objects passes a set threshold (25% in our implementation), it compacts the swap file in a heap task. MRT also cleans up after reclaimable and reclaimed objects by checking whether an object being freed is a stub; if so, it deletes the reclamation table entry corresponding to the stub. If an object is reclaimable, MRT deletes the memory-resident copy of the underlying object; if the object is reclaimed, MRT simply marks its saved copy in the swap file for deletion without needing to fault it in.

4.6.4 Design Tradeoffs and Alternatives

By tracking working sets and faulting in objects in software at the runtime level, Marvin achieves a clean design, albeit with some drawbacks. First, Marvin cannot reclaim objects accessed by native libraries: native libraries have no way to detect stubs and no recourse for faulting in reclaimed objects. Second, software working set estimation and object faulting add overhead, particularly to compiled code. We evaluate this overhead in Section 4.8.

Marvin moves almost all memory management into the runtime because we believe that the runtime's better access to information about app behavior makes it better suited for managing memory. The functionality remaining in the kernel is the minimum required by existing Linux kernel design; if Marvin was built on top of an exokernel [31, 67], it could move even more functionality into the runtime. A variety of other designs are possible that split functionality between the runtime and kernel in different ways.

Kernel-level working set estimation would reduce the overhead of accessing objects, but it would suffer from false sharing if an app's working set is mixed with unused objects across 4KB pages. Faulting in memory at the kernel level would similarly reduce object access overhead but would require more extensive re-design of the runtime garbage collector to avoid unnecessary swapping activity. By tying the granularity of memory management to

the size of pages, kernel-level memory management will also become inflexible as large pages become more common and the disparity between object sizes and page sizes widens. In any case, kernel-level memory management would require some sort of ahead-of-time swap mechanism to satisfy the latency requirements of modern mobile platforms (Figure 4.3), and even adding ahead-of-time swap to the existing Linux kernel would require significant implementation effort.

Marvin's garbage collector is different from the original bookmarking collector [52] in that it maintains exact reachability information with stubs rather than conservatively storing approximate reachability information. The latter approach requires the garbage collector to perform less work when evicting pages and scanning the heap, but it can result in the heap being needlessly occupied with dead objects.

4.7 Marvin Prototype

We implemented a prototype of MRT by modifying ART on Android 7.1.1 (which includes Linux 3.18.31). Our implementation includes a modified version of ART's ARM64 compiler, allowing our prototype to support Android devices with 64-bit ARM processors. Our changes to the ART codebase resulted in 3475 additional lines of code [103].

In addition to modifying ART, we made a small modification to the version of OpenJDK included with Android 7.1.1, namely, we added fields to the Object class definition to mirror the bytes added to the object header in ART. We also changed a source file in the Android framework (`ProcessList.java`) to increase a hard-coded limit on the number of concurrently running apps since Marvin is able to run more.

Our experiments require us to manually trigger reclamation, so we did not implement automated reclamation in the MK. However, our MRT implementation includes the reclamation table and performs all operations required to support kernel memory reclamation.

array-length	iget-*
instance-of	iput-*
check-cast	aget-*
invoke-interface	aput-*
invoke-virtual	

Table 4.1: DEX bytecode instructions for which Marvin performs object access interposition.

4.7.1 Object Access Interposition

We implemented MRT's object access interposition by adding specialized functionality to the ART interpreter and compiler. This lets MRT interpose on object accesses from both DEX bytecode running in the interpreter and compiled OAT code running natively. The following section describes in detail how we modified each component.

MRT interpreter. The ART interpreter internally represents each Java object as a C++ *mirror object*, which it manipulates when executing DEX bytecode instructions that read or write an object (shown in Table 4.1). The mirror object's type definition includes methods to read or write the data at a given offset within the object's memory footprint, and the interpreter code calls these methods when executing DEX instructions. To add object access interposition to the interpreter, we modified the mirror object methods to implement Marvin's features.

For example, to redirect object accesses through stubs and perform on-demand object faulting, we added a preamble macro to each mirror object method. The preamble first checks if the object is actually a stub. If so, it casts the `this` pointer to a stub, calls a method that locks the stub's reclamation table entry (RTE), checks the RTE's resident bit, and if the resident bit is cleared, calls a method to fault in the object from disk. The preamble then gets the address of the underlying object from the RTE and invokes the mirror object method on the underlying object. Finally, the preamble unlocks the RTE and returns the

result of the mirror object method, if any.

ART contains multiple interpreter implementations, and the default is the “mterp interpreter,” an interpreter written in assembly. When the mterp interpreter executes DEX instructions that read or write an array, it directly accesses the array’s memory, bypassing the mirror object methods. To allow Marvin to interpose on array accesses, we instead use the “switch interpreter,” an interpreter written in C++ that calls the mirror object methods when executing array accesses.

MRT compiler. Java code in Android framework libraries and portions of app Java code execute as native code, which is compiled by the ART compiler either statically after installation or dynamically with just-in-time (JIT) compilation. We added object access interposition to this compiled code by modifying the compiler’s assembler to generate additional assembly instructions that implement Marvin’s features when it performs code generation for object accesses. Since we used ARM64 devices for our testing and evaluation, we added support for object access interposition to the ARM64 assembler.

Each operation described above for the interpreter’s implementation of stub redirection and object faulting has a corresponding block of ARM64 instructions in compiled code. The main difference is that when a stub is detected, the compiled code must explicitly overwrite the register holding the stub’s address with the address of the underlying “real object;” it then loads the stub’s address back into that register when it is done with the object. In the common case, when an object is not a stub (or when it is, but its underlying “real object” is memory-resident), execution branches past many of the added object-faulting instructions.

4.7.2 Potential Optimizations

Our implementation of object access interposition in the MRT compiler is unoptimized, and the per-object-access overhead of compiled code could be reduced with deeper compiler integration. We modified the ARM64 assembler, which translates intermediate representation (IR) instructions to ARM64 binary code. Our implementation generates ARM64 instruc-

tions performing object access interposition for every IR instruction that reads or writes an object, even though many of those instructions only need to execute when the object's register allocation begins or ends. An optimized version of the compiler, with modifications at the IR level, could decrease both the execution and size overhead of compiled code.

The MRT compiler has other areas for optimization. For instance, we noticed situations where ARM64 parameter registers (`x0`–`x7` and `d0`–`d7`) appear to be live but are not reported as such by the ART compiler's `LocationSummary` class; therefore, we conservatively save all parameter registers to the stack when performing a procedure call. Saving only live parameter registers would further reduce code size overhead. In addition, the ARM64 assembler makes a maximum of two scratch registers available at any time, which required us to save some backpointers and add more instructions to juggle required state among the limited available registers.

4.8 Evaluation

Our evaluation demonstrated that Marvin successfully met its design goals. It could: (1) quickly reclaim memory on-demand; (2) maintain high memory utilization by sharing memory efficiently among apps rather than killing them; and (3) achieve the previous two goals with low overhead and no app changes.

4.8.1 Evaluation Setup

We ran our experiments on a Google Pixel XL smartphone with 4GB RAM and a quad-core Qualcomm Snapdragon 821 CPU. The smartphone ran either the open-source release of Android 7.1.1 (AOSP tag android-7.1.1_r57) or our Marvin implementation based on that release. Both our Marvin implementation and our baseline Android build included a change to the Android framework to increase a hard-coded cap on the number of concurrently running apps.

Our experiments used a mix of synthetic apps for microbenchmarking and real-world applications. We built several synthetic workloads that simulate various memory footprints

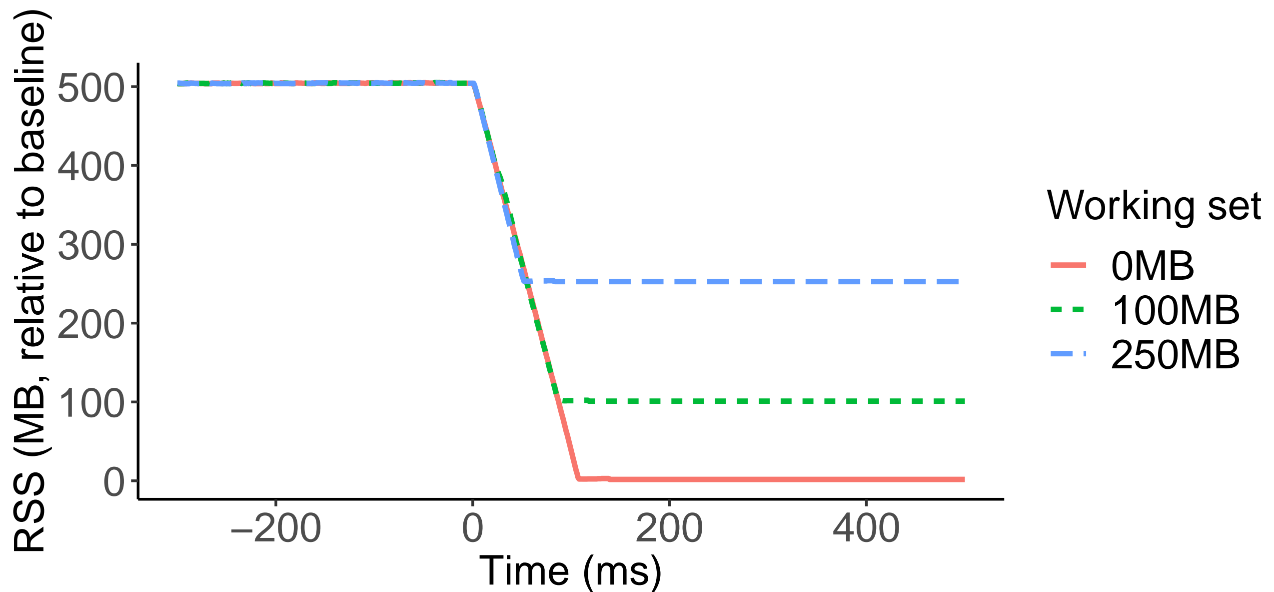

Figure 4.6: Memory usage as Marvin reclaims memory from a benchmark app with a 500MB heap and different working set sizes. Marvin took 108ms to reclaim 500MB, much faster than the nearly 8 seconds required by Android with Linux swap to allocate the same amount of memory.

and working set sizes to measure the effect of various working set sizes on Marvin compared to Android. We also used PCMark for Android, a commercial benchmark app based on real-world apps, to measure Marvin's overhead on real apps [11]. We chose two benchmarks from the test suite (Writing 2.0 and Data Manipulation) since the remaining three test the performance of native libraries.

4.8.2 Memory Reclamation

Marvin must be able to quickly reclaim memory from running apps when a new or existing app needs to allocate large amounts of memory. Its ahead-of-time swap mechanism ensures that each MRT instance has a pool of clean memory that can be quickly reclaimed without swapping to disk. In this section, we measure the latency of reclaiming memory for apps with different working set sizes. Reclamation latency depends on the app's working set size

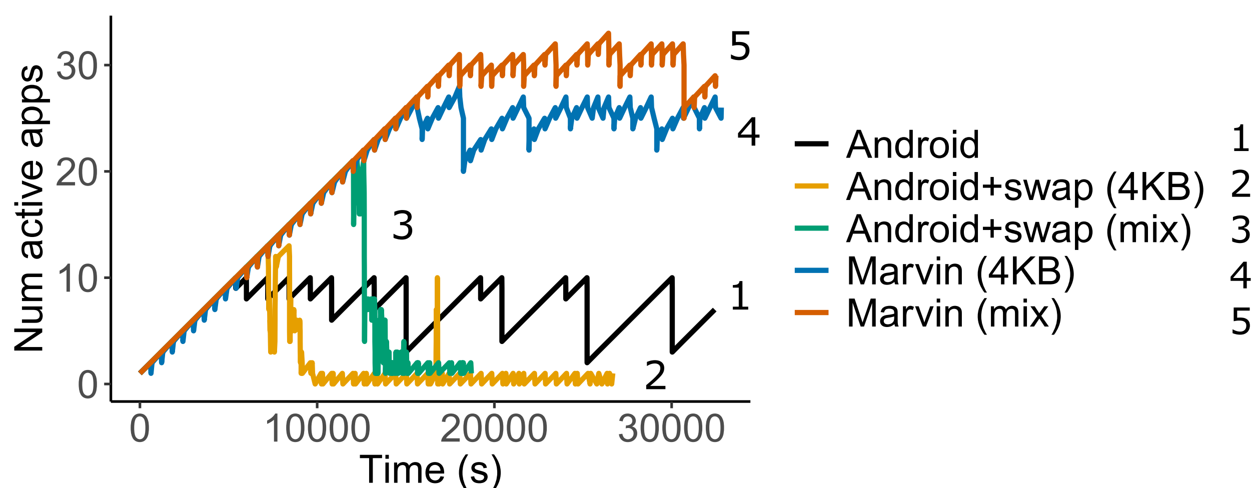

Figure 4.7: Count of active apps over time when starting instances of our benchmark app. Marvin runs more than twice as many apps as regular Android before needing to kill any apps; on Android with a swap file, most apps are alive but inactive due to constant swapping activity.

since Marvin can reclaim more memory from apps with smaller working set sizes.

For our prototype, we use `madvise` to return memory from MRT to the kernel. This design lets us trigger reclamation rather than waiting for memory pressure. Our MRT prototype reclaims memory when an app transitions to the background and then periodically while the app remains in the background.

Figure 4.6 shows memory usage over time for apps with a 500MB heap and differing working set sizes. RSS values shown are relative to the RSS reported for a minimal Android app with a single empty Activity (approx. 80MB). At time 0ms, MRT begins to return memory from the app to the OS. Marvin returned 250MB of memory in 52ms and 500MB of memory in 108ms. In comparison, as shown in Figure 4.3, Android with a Linux swap file took nearly 8 seconds to free and allocate 500MB of memory under memory pressure. Using ahead-of-time swap let Marvin reclaim memory over 60x faster than Android with Linux swap could allocate the same amount of memory, allowing Marvin to meet the strict latency requirements of mobile apps.

4.8.3 Memory Utilization

To demonstrate Marvin's more efficient memory manager, we ran multiple instances of an app with a large memory footprint and a limited working set, and we counted the number of *active* apps that were alive and making progress on their workloads. Each app had a 220MB heap filled with arrays, and it deleted and reallocated 20MB of those arrays every 5 seconds. We used two different heap compositions: one where the apps had heaps filled with 4KB arrays, and one where they had an even mix of 4KB and 1MB arrays (similar to the bimodal distribution of real apps in Figure 4.1). We consider an app "inactive" if it fails to perform a round of its workload for 20 seconds after the previous round; we consider it "active" once again if it succeeds in performing a round of its workload within 7 seconds of the previous round. We started a new app instance every 10 minutes to give Marvin time to perform background work. For unmodified Android, only the data for the apps with 4KB arrays is shown, because its behavior was nearly identical for the 4KB/1MB mix.

As shown in Figure 4.7, Marvin ran over 2x as many apps concurrently as unmodified Android and over 1.5x-2x as Android with a Linux swap file enabled; however, swapping left almost all apps unusable. While baseline Android begins killing apps when physical memory runs out, Android with swap keeps more apps alive. However, without a bookmarking garbage collector, the system experienced constant swapping activity, which prevented most apps from making progress on their workloads. Our experimental runs of Android with a swap file consistently ended early due to the device crashing.

Marvin made better use of device memory because it reclaimed unused memory from apps and used its bookmarking garbage collector to avoid touching that unused memory when running garbage collection. While Android only ran 10 apps concurrently, and Android with a swap file only briefly reached a maximum of 13 concurrent apps (4KB arrays) or 20 apps (4KB/1MB mix), Marvin ran 27 apps (4KB arrays) or 30 apps (4KB/1MB mix) concurrently. Marvin's memory reclamation and bookmarking garbage collector let it execute 1.5-2x as many apps concurrently while neither killing apps nor suffering performance degradation.

4.8.4 Runtime Overhead

While Marvin provides better memory management, it comes with a set of trade-offs. This section quantifies Marvin's four sources of overhead: (1) *execution time overhead* caused by Marvin's object access interposition in compiled OAT code; (2) *increased compiled code size* due to object access interposition; (3) *CPU utilization overhead* caused by Marvin's heap walks for working set estimation; and (4) *faulting overhead* when an app accesses a reclaimed object.

Execution time overhead of object access interposition. Native code produced by the MRT compiler has additional ARM64 instructions to support object access interposition. Some instructions (stub checks, dirty bit updates, and access-tracking bit updates) execute on every object access. Other instructions (indirecting object accesses through stubs and locking RTEs) execute only on accesses to reclaimable objects. We measured the overhead of the added instructions that apply to all object accesses using PCMark for Android, and we used a synthetic benchmark to illustrate the dependence of that overhead on the makeup of application code.

Figure 4.8 compares the performance of Marvin and unmodified Android on the PCMark benchmarks in our test set. Each bar shows the mean and standard deviation of five runs. We turned off stub creation and swapping when running PCMark, using the benchmarks to measure the overhead of the instructions added to every object access for stub checks and working set estimation. Marvin's score on the Writing 2.0 benchmark was nearly identical to Android's, and its score on the Data Manipulation benchmark was only 15% lower. These scores show that Marvin's overhead for accessing regular (i.e., non-reclaimable) objects is low for real-world apps.

Figure 4.9 explores the dependence of Marvin's overhead on the DEX instruction mix of application code. The graph shows Marvin's overhead executing a synthetic workload that performed a tunable proportion of object accesses (array reads and writes) and integer operations (addition and multiplication). Each point represents the mean and standard

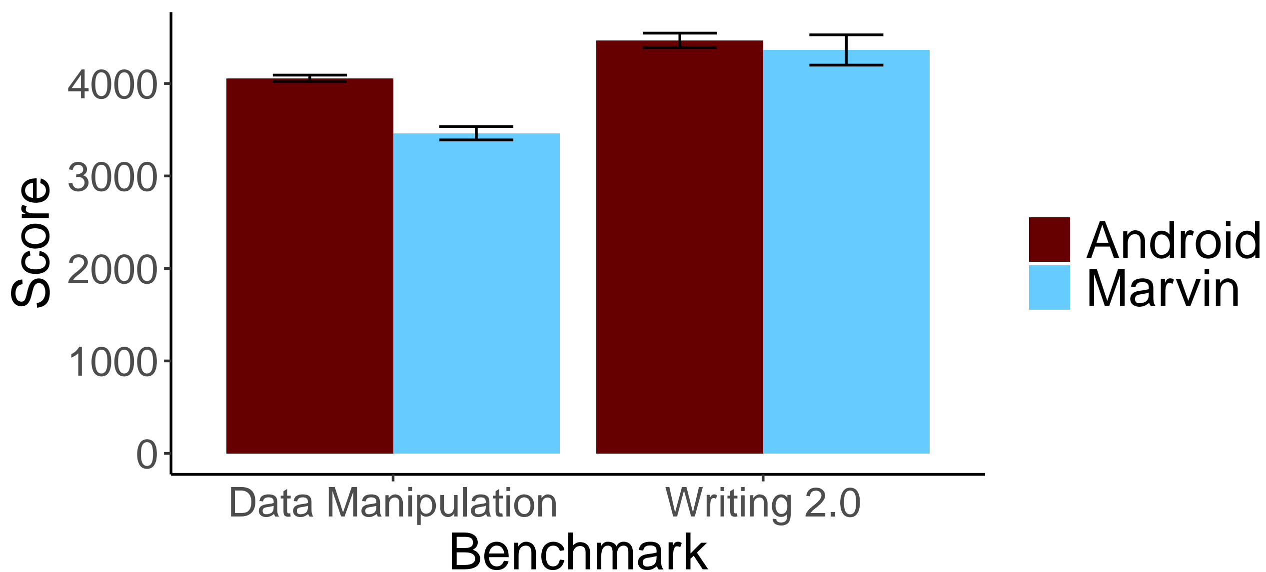

Figure 4.8: PCMark for Android benchmark results. Marvin's score is within 15% of Android's for both benchmarks, showing that Marvin adds low overhead for accessing non-reclaimable objects for real-world apps.

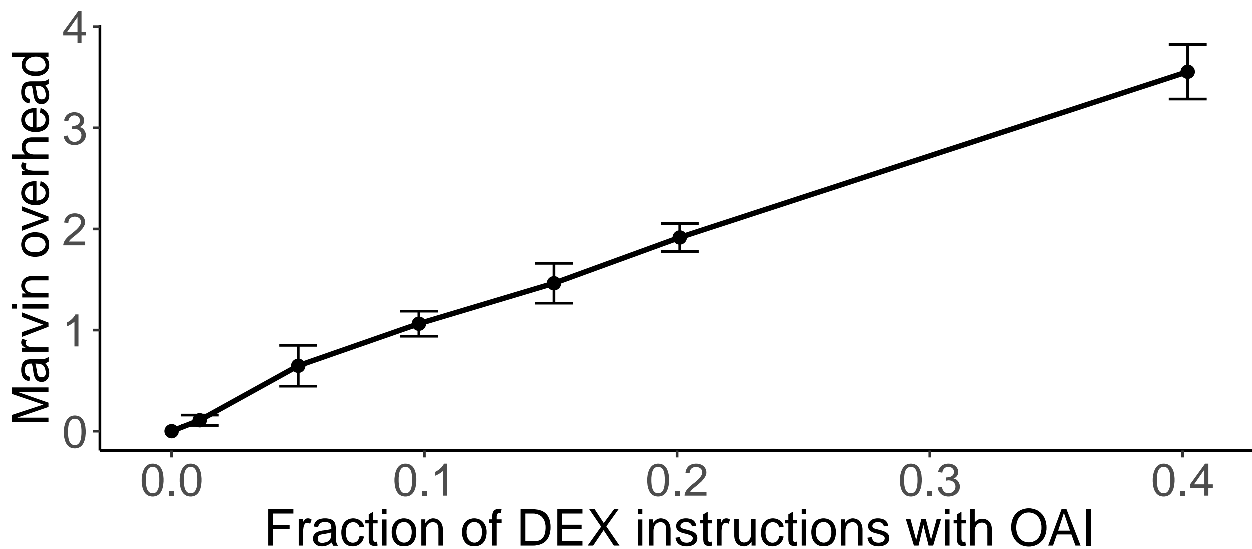

Figure 4.9: Overhead of Marvin for a synthetic workload with different proportions of DEX instructions with object access interposition (OAI). The point (0,0) represents the theoretical scenario of running no DEX instructions with OAI, while the other points show experimental results for the given workload mix. Marvin's overhead is lower when a smaller fraction of instructions have OAI.

deviation of Marvin's execution time overhead relative to Android for 40 iterations of the workload. For large proportions of object accesses, Marvin had relatively high overhead (e.g., 350% overhead for 40% object accesses), while for low proportions of object accesses, Marvin's overhead was minimal (e.g., 10% overhead for 1% object accesses). PCMark's 15% overhead indicates that the real app workloads represented by PCMark have low proportions of object accesses.

Although these overheads are already reasonable, they could be improved with optimizations. As noted in Section 4.7, deeper compiler integration would allow Marvin to reduce overhead by performing object access interposition less frequently.

Code size overhead of object access interposition. The ARM64 instructions added by Marvin's object access interposition also increase the size of compiled native code. To measure the increase in code size, we compared the compiled Android framework libraries generated by Marvin to the framework libraries on unmodified Android. Marvin increased the total size of the ARM64 framework libraries (in the `/system/framework/arm64` and `/system/framework/oat/arm64` directories on the Android filesystem) from 117 MB to 292 MB. This code size overhead, while relatively high, could be reduced significantly with deeper compiler integration (Section 4.7).

CPU utilization overhead of heap walks. Our Marvin prototype performs the heap walks required for working set estimation by invoking the concurrent garbage collector and piggybacking off its heap walk. Our prototype performs a heap walk every 5 seconds when an app is in the foreground and every 30 seconds for an app in the background. In theory, this periodic invocation of the garbage collector across multiple MRT instances could add CPU utilization overhead. In practice, when running multiple apps in the background, we found that the difference between Marvin's and unmodified Android's CPU utilization was negligible, likely because GC invocations were so infrequent for background apps.

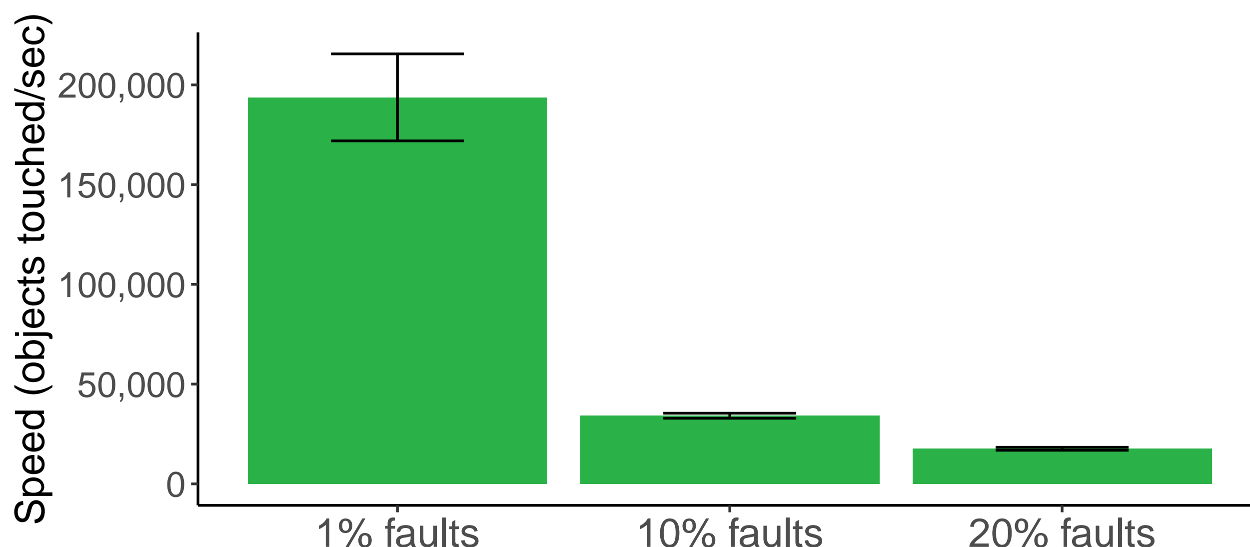

Figure 4.10: Speed of a benchmark app as it touches objects in its heap with different fractions of reclaimed objects. Object faulting never occurs in the user-visible foreground app due to Marvin's policies, but when Marvin does need to fault objects in from disk on-demand, its speed matches the expected result of trading off memory accesses for disk reads.

Overhead of faulting in objects. When an app first accesses a reclaimed object, Marvin must fault the object in from disk, adding significant latency to that initial access. Marvin's default policy eagerly restores all objects when an app moves to the foreground, so object-faulting latencies will never result in user-visible latency. Object faults may occur for background apps, but the Java working sets of apps in the background are generally quite small (less than 4MB for all commercial apps in our test set), so we expect object faulting to happen infrequently in practice.

We nonetheless studied the effect of object faulting on performance, to understand how Marvin might perform in situations where different policies or app workloads result in more object faulting. Figure 4.10 shows the effect of object faulting on a heap-walking benchmark app as it touches different fractions of reclaimed objects. The benchmark app looped over a set of 4KB arrays in its heap, reading five member variables of each array, and measured the speed at which it traversed the objects. Each bar shows the mean and standard deviation of

five measurements, and the disk cache on the device was cleared before each measurement. As expected, there was an inverse relationship between heap-walking speed and fraction of faults; for instance, speed dropped by 49% as the fraction of faults increased from 10% to 20%. This relationship is the natural result of exchanging low-latency memory accesses for high-latency disk accesses.

4.9 Related Work

Several recent systems provide swapping for mobile platforms but focus on page-granularity rather than object-granularity swapping. SmartSwap [112] predicts which apps are unlikely to be used and swaps out pages from those apps ahead-of-time. A2S [58] takes the opposite approach; it avoids swapping out pages from unused apps, since their pages will be freed anyways when they are terminated. MARS [51] optimizes Linux swapping to improve performance on flash storage devices. It disables garbage collection in background apps and reclaims memory from those apps. DR. Swap [111] proposes using NVRAM rather than flash storage to store swapped-out pages and satisfying reads by reading directly from NVRAM. Choi et al. [17] improve the performance of an in-memory file system by co-designing the swap mechanism to minimize I/O.

A significant body of work examines the issue of providing persistent memory for object-oriented languages [75, 85, 94, 6, 18, 106]. These systems checkpoint objects to disk or non-volatile memory, but they do so to ensure safety in the face of failures rather than swapping out unused memory. As a result, they focus on supporting transactional programming models that provide strong guarantees under failure [85, 18] and on implementing crash-safe garbage collection [18, 106] rather than on maximizing the number of apps that can run concurrently.

SSDAlloc [7] is a persistent memory system that, like Marvin, is motivated by the goal of helping apps with large memory footprints avoid memory pressure. Unlike Marvin's runtime-level object faulting and working set estimation, SSDAlloc allocates objects in separate virtual pages and uses the existing virtual memory system to estimate the working set and trigger its object fault handler.

Like Marvin, the bookmarking collector [52] aims to improve the performance of Java apps in memory-constrained environments. It assumes that the OS uses a traditional page-level swapping mechanism and focuses on letting the garbage collector run without unnecessary swapping. It conservatively stores approximate reachability information (bookmarks) that is used during garbage collection, whereas Marvin stores exact reachability information (stubs). The BMX garbage collector [35] similarly uses stubs to avoid expensive object accesses, but in the context of a distributed persistent object store.

Other recent work on garbage collection focuses on co-designing the GC and runtime to manage software caches more efficiently [81], co-designing the GC and virtual memory manager to improve performance [107], measuring the effect of GC on scalability [39], and designing GCs or memory managers for domains such as big data systems [79, 41].

With multiple runtimes performing their own memory management running on top of a single operating system, Android's architecture resembles the architecture of a virtual machine manager, where multiple guest operating systems run on top of a hypervisor. Marvin's runtime–OS cooperation is analogous to that between guest OS and hypervisor in the VMware ESX server [102], which uses a balloon driver to induce guest OSes to reclaim memory.

Wright et al. [105] present a system in which the architecture and Java runtime are co-designed for improved memory access performance. It features hardware modifications that allow an object-addressed CPU cache and an in-cache GC.

4.10 Conclusion

Users of mobile devices expect to use apps and switch between apps with low latency. As mobile apps have become more memory-hungry, device RAM capacities have not kept pace. Traditional desktop operating systems balance limited memory between multiple apps by swapping unused memory to disk, but classic swapping mechanisms cannot efficiently meet user latency expectations in the mobile setting. Many of the problems are due to the mismatch between the operating system's page-granularity memory management and modern,

object-oriented language runtimes: page-granularity working set estimation does not accurately capture app memory usage when objects are not page-sized, and garbage collection triggers unnecessary swapping activity. Swapping unused memory to disk on-demand also introduces excessive latency to the memory allocation critical path. Marvin overcomes these challenges by co-designing the memory manager with Android's Java runtime, performing much of the work of memory management in the language runtime rather than the operating system kernel. Marvin estimates working sets at the object granularity, checkpoints unused objects to disk ahead of time, and avoids unnecessary swapping during garbage collection. It implements these features using stubs for object reference indirection, a shared-memory reclamation table for coordination between the runtime and operating system, and compiler and interpreter modifications to interpose on object accesses by app code. As our experiments demonstrate, Marvin lets more apps run simultaneously and reclaims memory faster than Android using Linux swap while adding reasonable overhead.

Chapter 5
CONCLUSION

This thesis presented three systems that let distributed application developers easily manage data using high-latency resources. Many distributed apps allow users to interact with shared state, and they manage that interaction by coordinating between client and cloud components across wide-area networks. Diamond lets developers implement this shared state management with strong guarantees, simple synchronous interfaces, and scalable performance. Hercules improves on Diamond's shared state management by allowing real-time interactive apps to expose uncertainty in shared state and instantly respond to user input. Marvin focuses on client-side memory management, enabling mobile apps to take advantage of persistent storage without any additional developer effort.

5.1 Future Work

The area of systems that simplify distributed app development has vast potential, and the systems in this thesis are merely starting points. Technological advances in end-user devices, such as virtual reality headsets and haptic interfaces, will introduce new problems for distributed apps and tighten constraints on existing problems, making programming these apps that much harder. Changes to datacenter and edge technology will similarly introduce new problems but may also offer solutions: the "datacenter in a box" edge-computing devices envisioned by Satyanarayanan et al.'s tiered model of computing [92] would enable new distributed app topologies, with trusted infrastructure nodes that could be used for coordination located much closer to end-users than today's datacenters.

Even with existing platforms, the question of the ideal storage system interfaces for client app development is far from settled. These systems represent arguments for particular

interface designs that move the conversation forward, but they do not end that conversation by any means.

Shared data management This thesis presented Diamond and Hercules, two distributed storage systems focusing on making it easy for client-side developers to work with shared data. Diamond targeted traditional large-scale distributed apps, while Hercules targeted real-time interactive apps that are smaller-scale but have higher-fidelity user interaction. A natural line of work would be to explore the space between Diamond and Hercules. Traditional distributed apps and real-time interactive apps have much in common, and Hercules's ability to expose uncertainty would be useful to many traditional distributed apps as well. What would a version of Hercules that targets traditional distributed apps look like? Could we build a flexible system that supports both traditional distributed apps and real-time interactive apps, changing its behavior depending on the workload?

A related topic is that of adding more complicated backend architectures to Hercules. Hercules in its current form has a simple backend and places most of its technical emphasis on the interface between client and backend. Incorporating advances from recent datacenter distributed systems (and sacrificing Hercules's *Visible* view) would let it scale to more clients and larger shared data sizes. Would these changes be completely orthogonal to Hercules's client interface, or would it make more sense to add new views representing the additional stages involved in completing an operation?

A disparate but no less important area of inquiry is human-computer interaction research investigating what users expect and desire when working with shared state. The end goal of real-time interactive apps—making it seem like users around the world are in the same place, interacting with the same real-world objects—is fundamentally unattainable as long as the speed of light remains constant. These apps will always have to create illusions and make compromises. The questions of what exactly those illusions should look like, and which compromises they should make, are best informed by rigorously studying how real people interact with their apps and devices.

Memory management Marvin introduced runtime-level, object-granularity swapping, ahead-of-time swap, and a modified bookmarking garbage collector as its solution for mobile memory management. A natural follow-up would be to investigate other points in the design space, such as kernel-level, page-granularity swapping and Hertz et al.'s bookmarking collector [52] combined with Marvin's ahead-of-time swap. This alternative design could perform as well as Marvin or better, especially on native-memory-heavy workloads (such as apps with lots of images).

Another approach would be to stick with Marvin's object-granularity memory management and improve its performance with hardware co-design. Marvin uses current hardware with page tables and MMUs that perform page-granularity operations, so it has to perform most of its memory management in software, which adds significant overhead. Co-designing a mobile device's chipset with the language runtime, and implementing functionality such as object access interposition in hardware, would improve performance drastically.

Finally, future mobile platforms could provide persistent storage transparently, making mobile app development even easier. Marvin saves developers from having to manually implement caching, but it maintains Android's development model of threatening to kill apps at any time—Marvin just ensures that it only kills apps when their working sets actually overflow memory, rather than killing apps spuriously when they are not actively using memory. Developers must still manually persist data to ensure it survives across app deaths. A future platform could guarantee that all of an app's objects and operations are durable by default. As Bailey et al. note [8], emerging non-volatile memory technologies will make it easier to provide persistence by default, but providing that property introduces difficult interface design questions of its own.

BIBLIOGRAPHY

[1] David Abrahams and Stefan Seefeld. Boost.python – 1.72.0. https://www.boost.org/doc/libs/1_72_0/libs/python/doc/html/index.html.

[2] Michael Abrash. Latency – the *sine qua non* of ar and vr. http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/, 2012. Accessed 2019-6-10.

[3] Atul Adya, Gregory Cooper, Daniel Myers, and Michael Piatek. Thialfi: A client notification service for internet-scale applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 129–142, 2011.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 739–753, 2016.

[5] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: Managing datastore locality at scale with akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460, Carlsbad, CA, October 2018. USENIX Association.

[6] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, July 1995.

[7] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, 2011.

[8] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'11. USENIX Association, 2011.

[9] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[10] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*, pages 59–72, 2006.

[11] UL Benchmarks. Pcmark for android. `https://benchmarks.ul.com/pcmark-android`. Accessed: 2019-1-9.

[12] Kofi Amankwah Boamah. iphone on-board RAM, July 2017. `https://www.researchgate.net/figure/Phone-on-board-RAM-From-figure-8-it-is-clear-that-Apple-either-maintains-the-iPhon` `fig1_319307164`.

[13] Bumptech. Glide v4: Fast and efficient image loading for android. `https://bumptech.github.io/glide/`. Accessed: 2018-11-28.

[14] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol. In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP '15)*, pages 999–1052, 2015.

[15] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, 2006.

[16] Bytedeco. javacpp. `https://github.com/bytedeco/javacpp`.

[17] J. Choi, J. Ahn, J. Kim, S. Ryu, and H. Han. In-memory file system with efficient swap support for mobile smart devices. *IEEE Transactions on Consumer Electronics*, 62(3):275–282, August 2016.

[18] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[19] Kevin Conaway. Pyscrabble. `http://pyscrabble.sourceforge.net/`.

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.

[21] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *ACM MobiSys 2010*. Association for Computing Machinery, Inc., June 2010.

[22] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[23] Peter J Denning and Stuart C Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

[24] Android Documentation. Activity. `https://developer.android.com/reference/android/app/Activity`, 2018. Accessed: 2018-11-28.

[25] Android Documentation. Caching bitmaps. `https://developer.android.com/topic/performance/graphics/cache-bitmap`, 2018. Accessed: 2018-11-30.

[26] Android Documentation. Manage your app's memory. `https://developer.android.com/topic/performance/memory`, 2018. Accessed: 2018-11-28.

[27] Android Documentation. Saving ui states. `https://developer.android.com/topic/libraries/architecture/saving-states`, 2018. Accessed: 2018-11-28.

[28] Dormando. Memcached. `https://memcached.org/`. Accessed 2019-10-2.

[29] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[30] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[31] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, 1995.

[32] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[33] Facebook. Messenger. `https://www.messenger.com/`.

[34] Umar Farooq and Zhijia Zhao. Runtimedroid: Restarting-free runtime change handling for android apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 110–122, 2018.

[35] Paulo Ferreira and Marc Shapiro. Garbage Collection and DSM Consistency. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI '94)*, pages 229–241, Monterey CA, USA, United States, 1994.

[36] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[37] The Eclipse Foundation. Jetty – servlet engine and http server. `https://www.eclipse.org/jetty/`.

[38] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[39] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*, 2011.

[40] 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)*, pages 359–372, Santa Clara, CA, 2015.

[41] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015.

[42] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, and Irene Zhang. A.m.b.r.o.s.i.a: Providing performant virtual resiliency for distributed applications. Technical report, Microsoft, December 2018.

[43] Google. Add data to cloud firestore — firebase. `https://firebase.google.com/docs/firestore/manage-data/add-data`. Accessed on 2019-8-28.

[44] Google. Firebase. `https://firebase.google.com/products/firestore`. Accessed on 2019-7-31.

[45] Google. Firebase realtime database. `https://firebase.google.com/docs/database/`. Accessed: 2016-10-11.

[46] Google. Firebase realtime database — store and sync data in real time. `https://firebase.google.com/products/realtime-database/`. Accessed 2019-10-2.

[47] Google. Standalone toolchains. `https://developer.android.com/ndk/guides/standalone_toolchain`.

[48] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 93–106, 2012.

[49] J.N. Gray. Notes on data base operating systems. In R. Bayer, R.M. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*. Springer, 1978.

[50] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 169–184, Savannah, GA, 2016. USENIX Association.

[51] Weichao Guo, Kang Chen, Huan Feng, Yongwei Wu, Rui Zhang, and Weimin Zheng. Mars: Mobile application relaunching speed-up through flash-aware page swapping. *IEEE Transactions on Computers*, 65(3):916 – 928, March 2016.

[52] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 143–153, 2005.

[53] Todd Hoff. The architecture twitter uses to deal with 150m active users, 300k qps, a 22mb/s firehose, and send tweets in under 5 seconds. `http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html`, July 2013. Accessed 2019-10-2.

[54] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIX ATC '10, 2010.

[55] WhatsApp Inc. Whatsapp. `https://www.whatsapp.com/`.

[56] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[57] Tyler Kieft. Building a better instagram app for android. `https://instagram-engineering.com/building-a-better-instagram-app-for-android-c08f973662b`, 2014. Accessed: 2018-11-9.

[58] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. Application-aware swapping for mobile systems. *ACM Trans. Embed. Comput. Syst.*, 16(5s):182:1–182:19, September 2017.

[59] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, 2013.

[60] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[61] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[62] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, October 2006.

[63] John R. Lange, Peter A. Dinda, and Samuel Rossoff. Experiences with client-based speculative remote display. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX '08)*, pages 419–432, 2008.

[64] Niel Lebeck, Jonathan Goldstein, and Irene Zhang. Hercules: A multi-view cache for real-time interactive apps. Technical report, University of Washington, 2019.

[65] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. Technical report, University of Washington, 2019.

[66] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 151–165, 2015.

[67] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[68] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014.

[69] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical report, Massachusetts Institute of Technology, July 2012.

[70] Alessandro Margara and Guido Salvaneschi. We have a dream: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 142–153, 2014.

[71] Michelle Meyers. Android inches ahead of windows as most popular os. CNET, April 2017. `https://www.cnet.com/news/android-most-popular-os-beats-windows-statcounter/`.

[72] Microsoft. Interactive whiteboard for business – surface hub 2s — microsoft. `https://www.microsoft.com/en-us/surface/business/surface-hub-2`. Accessed 2019-8-12.

[73] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM.

[74] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, 2013.

[75] J. Eliot B. Moss. Design of the mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, April 1990.

[76] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 174–187, 2001.

[77] Mike Nakhimovich. Improving startup time in the nytimes android app. `https://open.blogs.nytimes.com/2016/02/11/improving-startup-time-in-the-nytimes-android-app/`, 2016. Accessed: 2018-11-9.

[78] Randy Nelson. The size of iphone's top apps has increased by 1,000% in four years. Sensor Tower, Jun 2017. `https://sensortower.com/blog/ios-app-size-growth`.

[79] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[80] Jakob Nielsen. Response times: The 3 important limits. `https://www.nngroup.com/articles/response-times-3-important-limits/`. Accesseds 2019-6-5.

[81] Diogenes Nunez, Samuel Z. Guyer, and Emery D. Berger. Prioritized garbage collection: Explicit gc support for software caches. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 695–710, New York, NY, USA, 2016. ACM.

[82] Oculus. Oculus rift s. `https://www.oculus.com/rift-s/`. Accessed 2019-8-12.

[83] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, 1988.

[84] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[85] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 161–174, New York, NY, USA, 1993. ACM.

[86] Parse Platform. Parse. `https://parseplatform.org/`. Accessed 2019-10-2.

[87] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 279–292, 2010.

[88] Ali Razeen, Alvin R. Lebeck, David H. Liu, Alexander Meijer, Valentin Pistol, and Landon Cox. Sandtrap: Tracking information flows on demand with parallel permissions. In *Proceedings of the 16th International Conference on Mobile Systems, Applications, and Services (MobiSys '18)*, 2018.

[89] Redis. Tutorial: Design and implementation of a simple twitter clone using php and the redis key-value store. `https://redis.io/topics/twitter-clone`.

[90] Anshu Rustagi. How we improved our android app "cold start" time by 28%. `https://redfin.engineering/how-we-improved-our-android-app-cold-start-time-by-28-a722e231314a`, 2018. Accessed: 2018-11-9.

[91] Salvatore Sanfilippo. Redis. `https://redis.io/`. Accessed 2019-10-2.

[92] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, pages 45–50, 2019.

[93] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[94] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In Antonio Albano and Ron Morrison, editors, *Persistent Object Systems*, pages 11–33, London, 1993. Springer London.

[95] Facebook Open Source. Fresco. `https://frescolib.org/`. Accessed: 2018-11-28.

[96] StackExchange. Creating and enabling an internal storage swap partition on rooted android kitkat. `https://android.stackexchange.com/a/89030`. Accessed: 2019-4-4.

[97] StackOverflow. ios app maximum memory budget. `https://stackoverflow.com/a/15200855`. Accessed: 2019-1-9.

[98] Mingshen Sun, Tao Wei, and John C.S. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, 2016.

[99] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012.

[100] 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 *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[101] Niraj Tolia, Jan Harkes, Michael Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST'04, pages 17–17, 2004.

[102] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[103] David A. Wheeler. SLOCCount, 2013. `http://www.dwheeler.com/sloccount/`.

[104] Wikipedia. Nim. `https://en.wikipedia.org/wiki/Nim#The_100_game`.

[105] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2005.

[106] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 70–83, New York, NY, USA, 2018. ACM.

[107] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Cramm: virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[108] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.

[109] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M. Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 723–738, Savannah, GA, 2016.

[110] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Viewbox: Integrating local file systems with cloud storage services. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 119–132, Santa Clara, CA, 2014.

[111] K. Zhong, X. Zhu, T. Wang, D. Zhang, X. Luo, D. Liu, W. Liu, and E. H.-M. Sha. Dr. swap: Energy-efficient paging for smartphones. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 81–86, Aug 2014.

[112] Xiao Zhu, Duo Liu, Kan Zhong, Jinting Ren, and Tao Li. Smartswap: High-performance and user experience friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 22:1–22:6, New York, NY, USA, 2017. ACM.