Consistency-Aware Durability

By

Aishwarya Ganesan

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2020

Date of final oral examination: October 2, 2020

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Aditya Akella, Professor, Computer Sciences
Michael M. Swift, Professor, Computer Sciences
Matthew D. Sinclair, Assistant Professor, Computer Sciences and
Affiliate, Electrical & Computer Engineering

Dedicated to my wonderful family
for their endless love, support, and encouragement

Acknowledgments

Many people have been instrumental to my successful completion of PhD. Many have guided, supported, and encouraged me before and throughout graduate school. In this section, I would like to express my heartfelt gratitude to them.

First and foremost, I would like to extend my deepest gratitude to my advisors, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. This PhD would not have been possible without their unwavering guidance and support during these five years of grad school. Andrea and Remzi made my PhD journey thoroughly enjoyable, inspiring me to pursue a career in research. Their weekly meetings are something I looked forward to each week. Not only these meetings gave me enough traction to complete projects on time but were also fun, mainly due to their enthusiasm for my work and partly due to their sense of humor.

Most of what I know about research, I have learned from Andrea and Remzi. Among other things, from Andrea, I learned how to organize my ideas when writing a paper. Her detailed and meticulous feedback on all the drafts that I have written has helped me become better at writing. Remzi taught me how to conduct experiments and present results. His emphasis on measuring one level deeper has helped me become a better researcher. Through their feedback on the various talks I gave, I learned how to present my work to others. I have also learned indirectly from

them; reading their book taught me a great deal about both operating systems and writing.

Andrea and Remzi care deeply about the growth of their students. Throughout my PhD, they helped me in many ways to become a better and independent researcher. They gave me the freedom to pick problems that I found interesting, encouraged me to apply for fellowships, created opportunities for me to mentor students, and finally, provided me with a chance to teach a graduate-level systems course. I am also very grateful to Andrea and Remzi for always being interested in my well-being, especially for their support during my pregnancy and postpartum. I honestly could not have asked for better advisors. Thank you, Andrea and Remzi!

I would like to express my deepest appreciation to my committee – Aditya Akella, Michael Swift, and Matthew Sinclair. I am grateful for their invaluable insights and suggestions for my research. I would also like to thank them for their interesting questions during the final defense. I would like to thank Mike for his detailed feedback on my dissertation and proposal drafts, which helped in improving this thesis to a great extent. I very much appreciate Mike and Aditya for writing letters of recommendation for me.

I am fortunate to have worked with a great set of colleagues at UW Madison – Ramnatthan Alagappan, Samer Al-Kiswany, Leo Prasath Arulraj, Youmin Chen, Yifan Dai, Tyler Harter, Jun He, Sudarsun Kannan, Jing Liu, Yuvraj Patel, Neil Perry, Thanumalayan Pillai, Anthony Rebello, Zev Weiss, Kan Wu, Yien Xu, and Suli Yang. I would also like to thank them for their insightful feedback on my research during the various group meetings and all the exciting and fun hallway discussions. I have been fortunate to collaborate with others at UW Madison and elsewhere whom I would like to thank. I had great pleasure of working with Aws Albarghouthi and Vijay Chidambaram on one of the projects. I have significantly benefited from my internship at Microsoft Research working

with my mentor Anirudh Badam. I also enjoyed working with Iyswarya Narayanan during my internship; discussions with her were immensely helpful.

Many others have helped me either directly or indirectly during my PhD. I would like to take this opportunity to express my appreciation to all of the CS staff and CSL for their help and support. Angela Thorp has been extremely helpful on numerous occasions, and I enjoyed working with her during GHC. I would also like to thank CloudLab for providing a great environment to run experiments. I would like to thank the Facebook fellowship program for generously supporting the last year of my PhD.

I am deeply thankful to my mentors at Microsoft Research India during the years before starting my PhD – Krishna Chintalapudi and Venkat Padmanabhan. I would not have applied to a PhD program if not for the unique research experience at MSR. I learned a great deal about research from Krishna and Venkat. I am incredibly grateful to them for their guidance and mentorship.

I would also like to extend my sincerest thanks to my friends in Madison. I would like to thank Lalitha for our fun times in Madison during my initial years of PhD and her support beyond. Numerous meetings and dinners with Harshad Deshmukh, Adalbert Gerald, Supriya Hirurkar, Rogers Jeffrey, Kaviya Lakshmipathy, and Meenakshi Syamkumar made my stay in Madison a fun-filled and memorable one. I am thankful for their friendship and support.

I cannot begin to express my thanks to my wonderful family, who have always been there for me, encouraged, and supported me throughout my life. I am immensely grateful to my extended family – my grandparents, aunts, uncles, and cousins. They have constantly cheered for me and taken pride in the smallest of my achievements. I am extremely lucky and fortunate to have their unparalleled support. I would also like to thank my parents-in-law, Leks Anna, Abi, and Thiru for their constant encour-

agement, support, and love. My mother-in-law has inspired me and reassured me on numerous occasions. I am highly indebted to Balamiss for her continuous support and encouraging me to always aim high.

I do not have enough words to thank my parents and sister for their unconditional love and boundless support. They have supported me every step of the way and encouraged me to pursue my passion. My parents nurtured my curiosity and instilled in me the critical values required in life. My mom managed to complete her graduate studies while being pregnant, which had motivated me many times in life, especially when I myself had a baby during my PhD. My mom stayed with me in Madison for several months taking care of my daughter while I was working on my research. For that and several other reasons, I am incredibly indebted to my mom. My dad has always been an inspiration to me. His hard work and dedication continue to motivate me. How he managed to grow to the highest ranks in his career almost starting from nothing amazes me. It is incredible how much my success and happiness mean to my younger sister, Akshaya. She has been a constant source of strength in my life. Without their relentless support, this PhD would not have been possible. Thank you, amma, appa, and Akshaya!

I would like to thank Aira, my wonderful daughter, for bringing me so much joy during the last year of PhD. You are a constant ray of light during all the hard times this year. Thank you for being kind and not being a difficult child.

Finally, one person that deserves my special thanks in many categories of this section is Ram – my partner, friend, mentor, and collaborator. More than eleven years ago when I was an undergraduate student in India, I worked on a project with Ram. Little did I know that years later, we would be married, have a kid together, and still be working on projects together. Ram is the person I go to for discussing my research ideas; our countless discussion sessions led to many of the ideas in this dissertation.

He is the first one to give feedback on my presentations and writing. His persistence, hard work, and attention to detail continue to amaze me. I am fortunate to have worked with Ram on the papers that form this dissertation, have co-taught a course with him at UW Madison, and above all, share the ups and downs of life and career with him. It would be an understatement to say that I would not have gotten through this PhD journey without him. Thank you, Ram!

Contents

Acknowledgments **ii**

Contents **vii**

List of Tables **xii**

List of Figures **xiv**

Abstract **xx**

1 Introduction **1**

1.1 Analysis of Modern Distributed Storage Systems 3

1.2 Building a Stronger and Efficient Durability Primitive . . . 7

1.3 Building Strong Consistency upon Consistency-aware Durability 9

1.4 Contributions 11

1.5 Overview 13

2 Background **15**

2.1 Distributed Storage Systems 15

2.2 Faults in Distributed Systems 17

2.2.1 Fail-stop Failures 17

2.2.2 Byzantine Faults 18

2.2.3 Storage Faults 18
2.3 Leader-based Majority Systems 21
2.4 Consistency Models 23
2.4.1 Strong Consistency 23
2.4.2 Weaker Models 24
2.5 Summary 25

3 Analysis of Distributed Systems Reactions to Storage Faults 26
3.1 Fault Model 28
3.2 Methodology 31
3.2.1 System Workloads 31
3.2.2 Fault Injection 31
3.2.3 Behavior Inference 34
3.3 System Behavior Analysis 35
3.3.1 Redis 36
3.3.2 ZooKeeper 41
3.3.3 Cassandra 45
3.3.4 Kafka 50
3.3.5 RethinkDB 55
3.3.6 MongoDB 58
3.3.7 LogCabin 60
3.3.8 CockroachDB 63
3.4 Observations across Systems 65
3.4.1 Systems employ diverse data integrity strategies . . 65
3.4.2 Faults are often undetected 67
3.4.3 Crashing is the most common reaction 67
3.4.4 Redundancy is underutilized 68
3.4.5 Crash and corruption handling are entangled 70
3.4.6 Local fault handling and global protocols interact in unsafe ways 74
3.4.7 Results Summary 76

3.5 File System Implications 77
3.6 Developer Interaction 78
3.7 Discussion 79
3.8 Summary and Conclusions 80

4 Building a Stronger and Efficient Durability Primitive 82
4.1 Durability Models 84
4.1.1 Immediate Durability 84
4.1.2 Eventual Durability 86
4.1.3 Consistency and Durability 89
4.2 Consistency-aware Durability: A New Durability Primitive 90
4.3 CAD Design 92
4.3.1 Leader-based Majority Systems 92
4.3.2 Failure Model and Guarantees 93
4.3.3 Update Path 94
4.3.4 State Durability Guarantee 95
4.3.5 Handling Reads: Durability Check 96
4.3.6 Read-triggered Durability 98
4.3.7 Correctness 99
4.4 Implementation 100
4.5 Evaluation 101
4.5.1 Write-only Micro-benchmark 103
4.5.2 YCSB Macro-benchmarks 104
4.5.3 Durability Guarantees 109
4.5.4 Summary 111
4.6 Implementing CAD in Redis 111
4.6.1 Redis Overview 112
4.6.2 Redis Implementation 112
4.6.3 Performance 113
4.7 Discussion 115
4.8 Summary and Conclusions 118

5 Building Strong Consistency upon Consistency-aware Durability **119**
5.1 Consistency vs. Performance 120
5.2 Stronger and Efficient Consistency with CAD 123
5.2.1 Cross-client Monotonic Reads and CAD 124
5.2.2 Utility of Cross-client Monotonic Reads 125
5.2.3 Need for Scalable Cross-client Monotonic Reads . . 127
5.3 ORCA Design 127
5.3.1 Guarantees 127
5.3.2 Cross-Client Monotonic Reads with Leader Restriction 129
5.3.3 Scalable Reads with Active Set 130
5.3.4 Active Set Membership using Leases 131
5.3.5 Correctness 134
5.3.6 Implementation 135
5.4 Evaluation 136
5.4.1 Read-only Micro-benchmark 136
5.4.2 YCSB Macro-benchmarks 137
5.4.3 Performance in Geo-Replicated Settings 139
5.4.4 ORCA Consistency 142
5.5 Application Case Studies 144
5.6 Summary and Conclusions 146

6 Related Work **148**
6.1 Corruption and Errors in Storage Stack 148
6.2 Storage Fault Injection 149
6.2.1 File-system Studies 149
6.2.2 Studies on Layers Above the File System 150
6.3 Analyzing Distributed System Reliability 151
6.3.1 Model Checkers and Bug Finding Tools 151
6.3.2 Generic Fault Injection 152

6.3.3 Bug Studies 152
6.4 Durability Semantics 153
6.5 Cross-client Monotonic Reads 154
6.6 Improving Distributed System Performance 156

7 Conclusions and Future Work **157**
7.1 Summary 158
7.1.1 Storage Faults Analysis 158
7.1.2 Consistency-aware Durability 159
7.1.3 Cross-client Monotonic Reads 161
7.2 Lessons Learned 161
7.3 Future Work 163
7.3.1 Storage Faults in Blockchain Systems 164
7.3.2 CAD for Other Systems 164
7.3.3 Transactions upon CAD and ORCA 166
7.3.4 Caching on CAD and ORCA 167
7.3.5 Active sets and Linearizability 168
7.4 Closing Words 168

Bibliography **171**

List of Tables

3.1 **Possible Faults and Example Causes.** *The table shows storage faults captured by our model and example root causes that lead to a particular fault during read and write operations.* 29
3.2 **Data Integrity Strategies.** *The table shows techniques employed by modern systems to ensure data integrity of user-level application data.* . . 66
3.3 **Scope Affected.** *The table shows the scope of data (third column) that becomes lost or inaccessible when only a small portion of data (first column) is faulty.* 69
3.4 **Outcomes Summary.** *The table shows the summary of our results. It shows the catastrophic outcomes caused by a single storage fault across all systems we studied. A cross mark for a system denotes that we encountered at least one instance of the outcome specified on the left.* 76
3.5 **Observations Summary.** *The table shows the summary of fundamental problems observed across all systems. A cross mark for a system denotes that we observed at least one instance of the fundamental problem mentioned on the left.* 77

4.1 **Immediate Durability Costs.** *The table shows the overheads of synchronous operations in Redis. The arrows show the throughput drop compared to the fully asynchronous configuration.* 85

4.2 **Durability.** *The table shows the durability-experiment results for the three durability models.* 110

5.1 **ORCA Correctness.** *The tables show how* ORCA *provides cross-client monotonic reads. In (a), weak-ZK and* ORCA *use asynchronous persistence; in (b), both replication and persistence are asynchronous.* 143

5.2 **Case Study: Location-tracking and Retwis.** *The table shows how applications can see inconsistent (non-monotonic), and consistent (old or latest) states with weak-ZK, strong-ZK, and* ORCA. 145

List of Figures

3.1 **CORDS Methodology.** *The figure shows an overview of our methodology to study how distributed systems react to local storage faults.* 32

3.2 ***Errfs* and Behavior Inference.** *The figure illustrates how errfs injects faults (corruptions and errors) into a block and how we observe the local behavior and the global effect of the injected fault.* 33

3.3 **Redis On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures of Redis. The logical structures take the following form: file_name.logical_entity. If a file can be contained in a single file-system block, we do not show the logical entity name.* 37

3.4 **Redis Behavior: Block Corruptions and Errors.** *The figure shows system behavior when corruptions (corrupted with either junk or zeros), read errors, write errors, and space errors are injected in various on-disk structures in Redis. Within each system workload (read and update), there are two boxes — first, local behavior of the node where the fault is injected and second, cluster-wide global effect of the injected fault. The rightmost annotation shows the on-disk logical structure in which the fault is injected. Annotations on the bottom show where a particular fault is injected (L – leader, F – follower). A gray box indicates that the fault is not applicable for that logical structure. For example, write errors are not applicable for any data structures in the read workload (since they are not written) and hence shown as gray boxes.* 38

3.5 **Redis Corruption Propagation.** *The figure depicts how the resynchronization protocol in Redis propagates corrupted user data in appendonly file (aof) from the leader to the followers, leading to a global user-visible corruption. Time flows downwards as shown on the left. The black portions denote corruption.* 39
3.6 **Redis Behavior: Bit Corruptions.** *The figure shows the behavior when bit corruptions are injected. For bit corruptions, we flip a* single bit *in a field within the on-disk structure. For example, appendonlyfile.db_num is part of appendonlyfile.metadata.* 40
3.7 **ZooKeeper On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures of ZooKeeper.* 42
3.8 **ZooKeeper Behavior.** *The figure shows system behavior when faults are injected in various structures in ZooKeeper.* 43
3.9 **ZooKeeper Write Unavailability.** *The figure shows how write errors lead to unavailability in ZooKeeper.* 44
3.10 **Cassandra On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures of Cassandra.* 45
3.11 **Cassandra Behavior: Block Corruptions and Errors.** *(a) and (b) show system behavior in the presence of block corruptions (corrupted with either junk (cj) or zeros(cz)), read errors (re), write errors (we), and space errors (se) when sstable compression is turned off and turned on, respectively.* 47
3.12 **Cassandra Corruption Propagation.** *The figure shows how the read-repair protocol in Cassandra propagates corrupted user data in a sstable file from a corrupted replica to other intact replicas.* 48
3.13 **Cassandra Behavior: Bit Corruptions.** *The figure shows the behavior in the presence of bit corruptions when sstable compression is off; the annotations on the bottom indicate the read quorum (R1 - quorum of 1, R3 - quorum of 3).* 49

3.14 **Kafka On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in Kafka.* 51
3.15 **Kafka Behavior: Block Corruptions and Errors.** *The figure shows system behavior in the presence of block corruptions and block errors.* 52
3.16 **Kafka Data Loss and Write Unavailability.** *The figure shows the scenario where Kafka loses data and becomes unavailable for writes due to a corruption.* 53
3.17 **Kafka Behavior: Bit Corruptions.** *The figure shows system behavior when bit corruptions are injected during a read workload.* 54
3.18 **RethinkDB On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in RethinkDB.* 55
3.19 **RethinkDB Behavior.** *The figure shows system behavior when faults are injected in various on-disk logical structures.* 56
3.20 **RethinkDB Data Loss.** *The figure shows the scenario where RethinkDB exhibits data loss due to a corruption.* 57
3.21 **MongoDB On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in MongoDB.* 58
3.22 **MongoDB Behavior.** *The figure shows the system behavior when faults are injected in various on-disk logical structures in MongoDB.* . . . 59
3.23 **LogCabin On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in LogCabin.* 61
3.24 **LogCabin Behavior.** *The figure shows system behavior when faults are injected in various on-disk logical structures.* 62
3.25 **CockroachDB On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in CockroachDB.* 63
3.26 **CockroachDB Behavior.** *The figure shows the system behavior when faults are injected in various on-disk logical structures.* 64

3.27 **Crash and corruption handling entanglement in Kafka.** *(a) shows how a crash during an update causes a checksum mismatch; in this case, the partially updated message is truncated. (b) shows the case where Kafka treats a disk corruption as a signal of a crash and truncates committed messages, leading to a data loss.* 71

4.1 **Asynchronous Persistence.** *The figure shows how an arbitrary data loss can occur upon failures with systems that persist asynchronously. Data items shown in grey denote that they are persisted (in the background).* . . 87

4.2 **Asynchronous Replication and Asynchronous Persistence.** *The figure shows how an arbitrary data loss can occur upon failures with systems that replicate and persist asynchronously.* 88

4.3 **CAD Update Path.** *The figure shows the update path in* CAD. *Data items that are durable are shown in grey boxes. In (i), the baseline performs both replication and persistence asynchronously; in (ii), the baseline synchronously replicates but persists lazily in the background. When a client writes item* b, *the write is acknowledged before* b *is made durable similar to the baselines.* CAD *then makes* b *durable in the background by replicating and persisting* b *on other nodes asynchronously.* 94

4.4 **CAD Durability Check.** *The figure shows how* CAD *works. Data items shown in grey are durable. In (i), the baseline is fully asynchronous; in (ii), the baseline synchronously replicates but asynchronously persists. At first, when item* a *is durable, read(*a*) passes the durability check. Items* b *and* c *are not yet durable. The check for read(*b*) fails; hence, the leader makes the state durable after which it serves* b. 97

4.5 **Write-only Workload: Latency vs. Throughput.** *The figure plots the average latency against throughput by varying the number of clients for a write-only workload for different durability layers.* 103

4.6 **YCSB Write Latencies.** *(a)(i) and (a)(ii) show the write latency distributions for the three durability layers for write-heavy YCSB workloads. (b)(i) and (b)(ii) show the same for read-heavy YCSB workloads.* 104

4.7 **YCSB Read Latencies: Write-heavy workloads.** *(a) and (b) show read latencies for eventual durability and* CAD *for write-heavy YCSB workloads. The annotation within a close-up shows the percentage of reads that trigger synchronous durability in* CAD. 106
4.8 **YCSB Read Latencies: Read-heavy workloads.** *(a) and (b) show read latencies for eventual durability and* CAD *for read-heavy YCSB workloads.* 107
4.9 **Overall Performance.** *The figure compares the throughput of the three durability layers. In (a), eventual and* CAD *are fully asynchronous; in (b), they replicate synchronously but persist lazily. The number on top of each bar shows the factor of improvement over immediate durability.* 108
4.10 **An Example Failure Sequence.** *The figure shows an example sequence generated by our test framework.* 109
4.11 **Redis Performance.** *The figure compares the throughput of immediate, eventual, and* CAD *durability layers in Redis. In (a), eventual and* CAD *synchronously replicate but asynchronously persist; in (b), they replicate and persist lazily. The number on top of each bar shows the performance normalized to that of immediate durability.* 114

5.1 **Linearizability.** *The figure shows possible values clients can observe upon reads in a linearizable system. The system has acknowledged updates* a_1, a_2, *and* a_3. *Time flows from left to right.* 121
5.2 **Causal Consistency.** *The figure shows possible values clients can observe upon reads in a causally consistent system. The system has acknowledged updates* a_1, a_2, *and* a_3; *updates* a_1 *and* a_2 *are causally related.* . . . 122
5.3 **Cross-client Monotonic Reads.** *The figure shows possible values clients can observe upon reads under our new consistency model. The system has acknowledged updates* a_1, a_2, *and* a_3. 124

5.4 **ORCA Guarantees.** *The figure shows possible values clients can observe upon reads with* ORCA*. The system has acknowledged updates* a_1*,* b_1*,* a_2*, and* b_2*. Once* $client_1$ *notices* b_2*, further reads must notice all updates upto* b_2*; thus, a later read from* $client_2$ *to item* a *notices* a_2*.* 128
5.5 **Non-monotonic Reads.** *The figure shows how non-monotonic states can be exposed atop* CAD *when reading at the followers.* 130
5.6 **Active Set and Leases: Unsafe Removal of Follower.** *The figure shows how if the leader removes a follower hastily then the system can expose non-monotonic states.* 131
5.7 **Active Set: Two-step Breaking of Lease.** *The figure shows how* ORCA *breaks leases in two steps.* 132
5.8 **ORCA Performance: Read-only Micro-benchmark.** *The figure plots the average latency against throughput by varying the number of clients for a read-only workload for the three systems.* 137
5.9 **ORCA Performance.** *The figure compares the throughput of the three systems across different YCSB workloads. In (a), weak-ZK and* ORCA *asynchronously replicate and persist; in (b), they replicate synchronously but persist data lazily. The number on top of each bar shows the performance normalized to that of strong-ZK.* 138
5.10 **Geo-distributed Experiment.** *The figure shows how the replicas and clients are located across multiple data centers in the geo-distributed experiment.* 139
5.11 **Geo-distributed Latencies.** *The figure shows the distribution of operation latencies across different workloads in a geo-distributed setting. For each workload, (i) shows the distribution of latencies for operations originating near the leader; (ii) shows the same for requests originating near the followers. The ping latency between a client and its nearest replica is <2ms; the same between the client and a replica over WAN is ~35 ms.* 140

Abstract

Modern distributed storage systems are emerging as the primary choice for storing massive amounts of critical data that we generate today. A central goal of these systems is to ensure data durability, i.e., these systems must keep user data safe under all scenarios.

To achieve high levels of durability, most modern systems store redundant copies of data on many machines. When a client wishes to update the data, the distributed system takes a set of actions to update these redundant copies, which we refer to as the system's *durability model*. At one end of the durability model spectrum, data is immediately replicated and persisted on many or all servers. While this immediate durability model offers strong guarantees, it suffers from poor performance. At the other end, data is only lazily replicated and persisted, eventually making it durable; this approach provides excellent performance but poor durability guarantees.

The choice of durability model also influences what consistency models can be realized by the system. While immediate durability enables strong consistency, only weaker models can be realized upon eventual durability. Thus, in this dissertation, we seek to answer the following question: *is it possible for a durability model to enable strong consistency guarantees, yet also deliver high performance?*

In the first part of this dissertation, we study the behavior of eight

popular modern distributed systems and analyze whether they ensure data durability when the storage devices on the replicas fail partially, i.e., sometimes return corrupted data or errors. Our study reveals that *redundancy does not provide fault tolerance*; a single storage fault can result in catastrophic outcomes such as user-visible data loss, unavailability, and spread of corruption.

In the second part, to address the fundamental tradeoff between consistency and performance, we propose *consistency-aware durability* or CAD, a new way to achieving durability in distributed systems. The key idea behind CAD is to shift the point of durability from writes to reads. By delaying durability upon writes, CAD provides high performance; however, by ensuring the durability of data before serving reads, CAD enables the construction of strong consistency models.

Finally, we introduce *cross-client monotonic reads*, a novel and strong consistency property that provides monotonic reads across failures and sessions. We show that this property can be efficiently realized upon CAD, while other durability models cannot enable this property with high performance. We also demonstrate the benefits of this new consistency model.

1 Introduction

Distributed storage systems are central to building modern services and applications that run in today's data centers. For instance, storage systems such as Cassandra, Redis, MongoDB, and ZooKeeper are at the heart of many online services including Internet search, e-commerce, ride-sharing, and social networking [47, 102, 133, 135, 173–175, 177].

A paramount goal of these storage systems is to ensure data *durability*, i.e., the system must keep user data safe under all circumstances. If a storage service loses data, the company loses its reputation and customers, and potentially many millions of dollars [33, 160, 180]. For the end-user, data loss is a nuisance at best and devastation at worst.

A common technique used by most distributed storage systems to provide high levels of durability is that of *redundancy*: instead of storing a single copy, data is redundantly stored on several servers. Thus, even if a few servers fail, data will not be lost and still can be accessed by applications.

Unfortunately, how a system updates the redundant copies (which we refer to as the system's *durability model*) presents an unsavory tradeoff between guarantees and performance. At one extreme lies the *immediate durability* model in which the distributed system replicates and persists a write on many replicas before acknowledging the clients. By replicating and persisting in the critical path, this model guarantees that data will not be lost even when many or all nodes crash and recover. However, such strong durability guarantees come at a high cost: poor performance.

Forcing writes to be replicated and persisted, even with performance enhancements such as batching, reduces throughput and increases latency dramatically.

At the other extreme is *eventual durability*: a distributed system only lazily replicates and persists a write, perhaps after buffering it in just one node's memory. By acknowledging writes quickly, high performance is realized, but this model can arbitrarily lose data in the presence of failures, providing weak durability guarantees.

The choice of durability model, in addition to determining the safety of data and performance, also influences another important aspect of a distributed system – its consistency model. Consistency models describe the guarantees a client has with regard to reads, given a set of previous writes from that client or other clients. Many models have been proposed, from (strong) linearizability at one end [70] to (weak) eventual consistency at the other [52], with many points in between [91, 94, 96, 169, 170, 176]. Strong consistency usually hides the fact that multiple copies exist, thereby making it easier for applications and programmers to reason about the system behavior. Strong consistency is often achieved by employing immediate durability. For example, to prevent stale reads, a linearizable system (such as LogCabin [92]) synchronously makes writes durable; otherwise, an acknowledged update can be lost, exposing stale values upon subsequent reads.

In contrast, systems that employ eventual durability (such as Redis [134]) can only realize only weak consistency models. These systems can arbitrarily lose data upon failures, exposing stale and out-of-order data, and thus provide confusing semantics to applications.

Thus, distributed storage systems must choose between two unsavory options: offer strong guarantees but pay a high performance cost, or deliver high performance but settle for weak guarantees. Most modern systems favor performance over correctness, and thus adopt eventual dura-

bility; in fact, eventual durability is the default in many widely used systems [104, 136, 137]. Consequently, these systems offer high performance but only offer poor durability and consistency guarantees.

Thus, in this dissertation, we ask the following question. *Is it possible for a distributed system to provide strong durability and consistency while also delivering high performance?* We answer this question in the affirmative by introducing *consistency-aware durability* or CAD, a new way of achieving durability in distributed systems. The key idea behind CAD is to shift the point of durability from writes to that of *reads*: data is guaranteed to be durable before it is served upon reads. By delaying durability of writes, CAD achieves high performance; however, unlike eventual durability that can arbitrarily lose data, CAD guarantees that read data will never be lost and thus enables strong consistency models.

This dissertation has three parts to it. In the first part, we study the behavior of eight widely used modern distributed storage systems and analyze whether these systems ensure data durability in the presence of failures. We focus on the case where the disks on the replicas sometimes return corrupted data or errors upon accesses (§1.1). Second, to address the fundamental tradeoff between durability and consistency, and performance, we introduce CAD (§1.2). We design and implement CAD for leader-based majority systems and show its efficacy. Finally, we introduce *cross-client monotonic reads*, a novel, strong consistency property that can be realized efficiently upon CAD (§1.3). We also demonstrate the benefits of this new consistency property.

1.1 Analysis of Modern Distributed Storage Systems

Each replica in a distributed storage system depends upon its local storage stack to store data. The local storage stack is immensely complex consist-

ing of storage devices, firmware, and many layers of software including local file systems, device drivers, I/O schedulers, etc. Upon problems in any of these layers, the storage stack can return corrupted data or errors to file systems and applications above them [22, 24, 25, 97, 112, 124, 155, 156]. We refer to such corruptions and errors as storage faults.

Previous studies [26, 130, 187] have shown how storage faults are handled by local file systems such as ext3, NTFS, and ZFS. File systems, in some cases, simply propagate the faults as-is to applications; for example, ext4 returns corrupted data as-is to applications if the underlying device block is corrupted. In other cases, file systems react to the fault and transform it into a different one before passing onto applications; for example, btrfs transforms an underlying block corruption into a read error. Thus, in either case, distributed storage systems can encounter faults when running atop local file systems.

The behavior of modern distributed storage systems in response to storage faults is critical and strongly affects cloud-based services. Despite this importance, little is known about how distributed storage systems react to storage faults and whether these systems preserve the durability of data in the presence of storage faults.

A common and widespread expectation is that redundancy in higher layers (i.e., across replicas) enables recovery from local storage faults [30, 50, 68, 82, 156]. For example, an inaccessible block of data in one node of a distributed storage system would ideally *not* result in a user-visible data loss because the same data is redundantly stored on many nodes. Given this expectation, in the first part of the thesis, we answer the following questions: *How do modern distributed storage systems behave in the presence of local storage faults? Do they use redundancy to recover from storage faults on a single replica?*

To answer these questions, we build a fault-injection framework called CORDS which includes the following key pieces: *errfs*, a user-level FUSE file

system that systematically injects storage faults, and *errbench*, a suite of system-specific workloads that drives systems to interact with their local storage. For each injected fault, CORDS automatically observes resultant system behavior. We studied eight widely used systems using CORDS: Redis [134], ZooKeeper [16], Cassandra [13], Kafka [15], RethinkDB [143], MongoDB [101], LogCabin [92], and CockroachDB [41].

The most important overarching lesson from our study is that redundancy does not provide fault tolerance in many distributed storage systems: a single storage fault in only one of the replicas can induce catastrophic outcomes. Despite the presence of checksums, redundancy, and other resiliency methods prevalent in distributed storage, a single fault can lead to data loss, corruption, unavailability, and, in some cases, the spread of corruption to other intact replicas. Because distributed storage systems inherently store redundant copies of data, and we inject only one fault at a time, these behaviors are surprising and undesirable.

The benefits of our systematic study are twofold. First, our study has helped us uncover numerous bugs in eight widely used systems. We find that these systems can silently return corrupted data to users, lose data, propagate corrupted data to intact replicas, become unavailable, or return an unexpected error on queries. For example, a single write error during log initialization can cause write unavailability in ZooKeeper. Similarly, corrupted data in one node in Redis and Cassandra can be propagated to other intact replicas. In Kafka and RethinkDB, corruption in one node can cause a user-visible data loss.

Second, more importantly, our study has enabled us to make several observations across all systems concerning storage fault handling. We find that while some of the above undesirable outcomes are caused due to implementation-level bugs that could be fixed by moderate developer effort, most of them arise due to a few fundamental root causes in storage fault tolerance common to many distributed storage systems. We list

these root causes below.

- **Faults are often undetected locally.** While a few systems carefully use checksums, others completely trust lower layers in the stack to detect and handle corruption. Thus, we find that corruptions are often locally undetected in some systems. We also find that I/O errors are often not handled properly in many systems. More importantly, we find that locally undetected faults lead to immediate harmful global effects (e.g., spread of corruption).

- **Crashing is the most common reaction.** Even when systems reliably detect faults, in most cases, they simply crash instead of using redundancy to recover from the fault.

- **Redundancy is underutilized.** Although distributed storage systems replicate data and functionality across many nodes, a single storage fault on a single node can result in harmful cluster-wide effects. Surprisingly, many distributed storage systems do not consistently use redundancy as a source of recovery.

- **Crash and corruption handling are entangled.** Systems often conflate recovering from a crash with recovering from corruption, accidentally invoking the wrong recovery subsystem to handle the fault. This conflation ultimately leads to poor outcomes such as data loss.

- **Local fault handling and global protocols interact in unsafe ways.** Local fault-handling behaviors and global distributed protocols such as read repair, leader election, and re-synchronization, sometimes interact in an unsafe manner. This unsafe interaction often leads to propagation of corruption to intact replicas or data loss.

To summarize, our study indicates that storage faults can have adverse effects on durability in most distributed storage systems: a single storage

fault can lead to loss of data, system unavailability, or spread of corruption. Most outcomes result due to a few fundamental problems concerning storage fault tolerance prevalent across many systems. Finally, these systems are not equipped to effectively use redundancy across replicas to recover from local storage faults.

1.2 Building a Stronger and Efficient Durability Primitive

In the second part of this thesis, we examine the fundamental tradeoff between strong guarantees and performance in distributed storage systems. When a user wishes to store a piece of data, the distributed storage system orchestrates a sequence of steps to make the data durable. We refer to this set of actions a system takes as its *durability model*. In particular, durability models describe how a distributed system replicates and persists data across machines. A system's underlying durability model has strong implications on both consistency and performance.

As we noted earlier, at one end of the spectrum, immediate durability synchronously persists data on many nodes, and thus offers strong guarantees; however, it is too slow. At the other end lies eventual durability which only asynchronously persists data, thereby offering high performance; however, it does so by trading off strong durability and consistency.

As a result, practitioners are left with the unsavory decision of choosing strong guarantees or performance but not both. Given the poor performance of immediate durability, many deployments prefer eventual durability [74, 106, 125]. In fact, eventual durability is the *default* in many popular systems (e.g., Redis, MongoDB). Even when using systems that are immediately durable by default (e.g., ZooKeeper), practitioners disable synchronous operations for performance [55]. Therefore, these deploy-

ments and systems settle for weaker consistency and durability guarantees.

Thus, we examine whether it is possible for a durability layer to enable strong consistency, yet also deliver high performance. We first note that to provide high performance, the system cannot employ synchronous writes because it is simply too expensive. Second, we realize that what clients observe upon reads is important for most consistency models. Based on these two insights, we rethink the durability layer and propose *consistency-aware durability* or CAD. The key idea behind CAD is to shift the point of durability from writes to reads; data is replicated and persisted before it is *read*. By delaying durability of writes, CAD achieves high performance. However, by making data durable before it is read, CAD guarantees that data that has been read by clients will never be lost and can be recovered even after failures; this enables strong consistency across failures, as we show in the next part of the thesis.

CAD does *not* incur overheads on every read; for many workloads, data can be made durable in the background before applications read it. While enabling strong consistency, CAD does not guarantee complete freedom from data loss; a few recently written items that have not been read yet may be lost if failures arise. However, given that many widely used systems adopt eventual durability and thus settle for weaker consistency [104, 136, 137], CAD offers a path for these systems to realize stronger consistency and durability guarantees without compromising on performance.

We implement CAD in ZooKeeper [16], a leader-based majority system. Our experiments show that ZooKeeper with CAD is significantly faster than immediately durable ZooKeeper (optimized with batching) while approximating the performance of eventually durable ZooKeeper for many workloads. Even for workloads that mostly read recently written data, CAD's overheads compared to eventually durable ZooKeeper are

small (only 8%). Through rigorous fault injection, we demonstrate the robustness of CAD's implementation in ZooKeeper; CAD ensures the durability of data that has been read by clients in hundreds of crash scenarios. We also demonstrate that the consistency-aware durability idea applies to other systems as well by implementing CAD in Redis; we also present a performance evaluation of this implementation.

1.3 Building Strong Consistency upon Consistency-aware Durability

In the last part of the thesis, we show how to realize strong consistency upon consistency-aware durability (CAD). Linearizability is the strongest guarantee that can be provided by a non-transactional distributed system [70, 120]. A linearizable system never allows clients to read stale data. Further, it prevents clients from seeing out-of-order states: the system will not serve a client an updated state at one point and subsequently serve an older state to any client. To provide such strong guarantees on reads, a linearizable system must pay the cost of immediate durability during write operations [87, 120]. In addition to using immediate durability, most linearizable systems restrict reads to the leader [80, 108, 119]. Such restriction limits read throughput and prevents clients from reading from nearby replicas, increasing latency.

In contrast, models such as causal consistency, monotonic reads, and eventual consistency perform well when compared to linearizability. These models are often built upon a weakly durable substrate and allow reads at many nodes. However, these models offer only weak guarantees. For example, consider the above two consistency properties: clients can never see stale or out-of-order data; systems that employ weaker models violate these two properties.

We note that preventing staleness requires expensive immediate dura-

bility upon every write. However, preventing out-of-order states can be useful in many scenarios and can be realized efficiently. The well-known monotonic reads consistency model [169, 170] seemingly prevents out-of-order states but is quite limited in this regard. In particular, while it avoids out-of-order reads within a *single* client session, it does not guarantee in-order states across clients or even different sessions of the same client application. In contrast, we introduce a new consistency model that provides much stronger guarantees; we refer to this new model as *cross-client monotonic reads*. Cross-client monotonic reads guarantees that a read from a client will return a state that is at least as up-to-date as the state returned to a previous read from *any* client, irrespective of failures and across sessions. We show that this property can be realized with high performance upon CAD. Without CAD, it is hard (if not impossible) to realize cross-client monotonicity efficiently. Specifically, immediate durability can enable it but is too slow; on the other hand, it simply cannot be realized upon eventual durability. Among the existing consistency models, only linearizability provides cross-client monotonic reads, albeit, at the cost of performance.

We design and build ORCA by implementing cross-client monotonic reads upon CAD in ZooKeeper. It is straightforward to provide cross-client monotonicity upon CAD when restricting reads to the leader (which limits read scalability). However, ORCA offers cross-client monotonicity while allowing reads at many replicas. To allow reads at many nodes while maintaining monotonicity, ORCA employs a lease-based active set technique and a two-step lease-breaking mechanism to correctly manage active-set membership. By permitting reads at many nodes, ORCA achieves low-latency reads by allowing clients to read from nearby replicas, making it particularly well-suited for geo-distributed settings. Further, ORCA can be beneficial in edge-computing use cases, where a client may connect to different servers over the application lifetime (e.g., due to mobility [132]),

but still can receive monotonic reads across these sessions.

We experimentally show that ORCA offers significantly higher throughput (1.8 – 3.3×) compared to strongly consistent ZooKeeper (strong-ZK). In a geo-distributed setting, by allowing reads at nearby replicas, ORCA provides 14× lower latency than strong-ZK in many cases. ORCA also closely matches the performance of weakly consistent ZooKeeper (weak-ZK). We show through rigorous tests that ORCA provides cross-client monotonic reads under hundreds of failure sequences generated by a fault-injector; in contrast, weak-ZK returns non-monotonic states in many cases. We also demonstrate how the guarantees provided by ORCA can be useful in two applications: social-media timeline and location-sharing.

1.4 Contributions

We list the main contributions of this dissertation.

- **CORDS Tool.** We design and build CORDS, a fault-injection framework to analyze how distributed systems react to storage faults. The framework consists of a user-level FUSE file system to inject storage faults into applications running upon local file systems, and a suite of system-specific workloads. Our framework is publicly available [1].

- **Vulnerabilities in Widely Used Systems.** We present a detailed behavioral study of eight widely used distributed systems on how they react to storage faults and uncover many previously unknown vulnerabilities in these systems. We have contacted developers of seven systems and five of them have acknowledged the problems we found. The vulnerabilities we reported can be found here [1].

- **Fundamental Observations about Storage Fault Tolerance.** We present a set of fundamental problems in storage fault tolerance that are

common to many modern distributed systems. For instance, we identify that many systems conflate corruptions that arise due to system crashes and storage corruptions, leading to undesirable outcomes such as data loss. Some of the lessons from this study have proved instrumental to building replicated state machines that correctly recover from storage faults [10] (our follow-on work, not a part of this thesis).

- **Durability Models.** We analyze how many widely used distributed storage systems make data durable upon writes. We identify the immediate and eventual durability models popular in these systems. We show how a distributed system's underlying durability model affects its consistency and performance characteristics.

- **Consistency-aware Durability.** We design consistency-aware durability (CAD), a new durability model for distributed systems, that shifts the point of durability from writes to reads, providing strong guarantees and excellent performance.

- **Cross-client Monotonic Reads.** We introduce cross-client monotonic reads, a new consistency model that provides strong guarantees without requiring expensive synchronous durability. Among the existing consistency models, only linearizability provides cross-client monotonicity; however, it is slow because it requires synchronous durability.

- **ORCA.** We design and implement cross-client monotonic reads and CAD for leader-based majority systems in a system called ORCA by modifying ZooKeeper. We present a rigorous evaluation of CAD and ORCA. We also implement and evaluate CAD in Redis.

1.5 Overview

We briefly describe the contents of the different chapters in the dissertation.

- **Background.** Chapter 2 provides background on distributed storage systems and the common failure models. We then discuss leader-based majority systems and different consistency models.
- **Analysis of Distributed Systems Reactions to Storage Faults.** Chapter 3 presents our study on how eight widely used distributed storage systems react to storage faults. We describe our fault-injection methodology and present the behavior analysis of different systems. We then present high-level observations related to storage fault tolerance that are common to systems we study.
- **Building a Stronger and Efficient Durability Primitive.** In Chapter 4, we analyze existing durability models and introduce consistency-aware durability or CAD, a new durability primitive that delivers high performance and facilitates construction of stronger consistency guarantees. We then present the design, implementation, and evaluation of CAD.
- **Building Strong Consistency upon Consistency-aware Durability.** In Chapter 5, we build cross-client monotonic reads, a new consistency model that provides strong guarantees and can be realized efficiently upon CAD. We describe ORCA, our design for leader-based systems. We then evaluate ORCA and demonstrate ORCA's utility for applications.
- **Related Work.** In Chapter 6, we discuss other research work and systems that are related to this dissertation. We first discuss work related to prevalence of storage faults, storage fault injection, and

studies on distributed systems reliability. We then compare our new durability primitive and consistency model with existing work and finally discuss other efforts to improving distributed system performance.

- **Conclusions and Future Work.** Chapter 7 summarizes this dissertation. We also present some lessons learned during the course of this dissertation and discuss possible directions our work could be extended.

2 Background

In this chapter, we provide a background on various topics relevant to this dissertation. We start with a brief overview of distributed storage systems (§2.1). Then, we discuss the various failures that are possible in distributed systems (§2.2) with an emphasis on storage faults (§2.2.3). We then briefly describe leader-based majority systems (§2.3). Finally, we discuss different consistency models that are related to this dissertation (§2.4).

2.1 Distributed Storage Systems

A distributed storage system consists of many servers. The servers may be located within a single data center or spread across multiple data centers. The storage system maintains some persistent state; this state can be a collection of key-value pairs, a database, etc. Clients interact with the system through a set of operations. The operations can vary across systems, but broadly operations can be classified into read-only (that retrieve data and do not modify any state) and write (that update existing or store new data).

These storage systems must meet a few important goals. First, the system must ensure *durability* of user data: it must not lose data that the clients entrust the system with. Second, the system must be highly *avail-*

able and be able to provide access to user data even in the presence of failures.

A key to building a system that offers the above properties is to employ redundancy: instead of having a single copy of data, the system maintains multiple copies of user data on disks of multiple machines. Thus, even if one machine fails, the data is not lost because copies of the same data exist on other machines. It is not just the data that is replicated but also the functionality; identical copies of a program run on multiple machines as well. Therefore, even if one machine fails, the code running on other machines keeps the system available.

Given that there are many copies of data, one key challenge that the distributed system faces is to maintain *consistency*, i.e., it must keep the copies of data identical with each other. An ideal distributed system must hide from clients that multiple copies exist and provide the illusion of a single copy. However, several challenges must be solved to achieve such ideal behavior. One challenge is dealing with failures. In the next section, we discuss the various failures that are possible in a distributed system (§2.2). We will then briefly discuss how existing systems maintain identical copies in the presence of failures (§2.3). Some distributed systems, for high performance and availability, choose not to provide the illusion of a single copy; we will discuss some of the possible (weaker) guarantees that can be provided by such systems (§2.4).

In addition to the above properties, a distributed storage system must also be *performant* and *scalable*. Scalability is usually achieved via sharding. Massive data is partitioned into many smaller units called shards, and the shards are distributed across many nodes; each shard fits within a single node. For durability and availability, each shard is in turn replicated. To update data across multiple shards, systems typically use transactional mechanisms. In this dissertation, we focus mainly on applying our ideas in non-sharded settings and therefore, we do not discuss shard-

ing in detail. However, as we discuss later (§4.7), our ideas can be applied as-is or extended to multiple shards.

2.2 Faults in Distributed Systems

In this section, we discuss the different types of failures that a distributed storage system must handle.

2.2.1 Fail-stop Failures

Under the fail-stop failure model, a component stops operating entirely. For example, a process might fail or stop running because of a power loss or an operating system crash. A network partition is also an example of fail-stop failure where a node on the network is not reachable to other nodes. Additionally, network packets may be lost, reordered, or delayed.

Before a component fails, it usually does not inform the other components in the system of the imminent failure. Thus, the other components have to detect that a component has failed. This detection is usually done using the mechanisms of *heartbeats* and *timeouts*. At a high level, the mechanism works as follows. A node in the distributed system periodically sends a heartbeat message to other nodes. If a node does not receive a heartbeat message for a period of time from another node, then the node considers the other node as failed. In such a case, there are two possibilities. Either the node might have crashed, or the node is alive, but its messages cannot reach the other node because of a network problem (e.g., a partition).

A failed node may recover after some time. For this reason, this model is also referred to as the fail-recover model. For example, a node that crashed due to a power loss recovers when power is restored afterwards. When a node crashes due to a power failure and later recovers, all the contents in its memory (DRAM) and disk caches are lost; only the persis-

tent data that is on the disk platter or solid-state drive survives. In case of a network partition, while a node cannot communicate with other nodes, its in-memory state usually remains intact. Systems that provide strong guarantees require $2f + 1$ nodes to tolerate f fail-stop failures; these systems are available only when the total number of fail-stop failures do not exceed f.

2.2.2 Byzantine Faults

In this failure model, components in the distributed system may behave arbitrarily. For example, a server might perform a computation incorrectly due to a malicious attack. A malicious node in the system might provide conflicting information to different parts of the system [86]. The node might even behave correctly at times that makes it appear like a functioning node (for example, serve correct data). Therefore, these faults are difficult to detect and tolerate. Protocols that tolerate such Byzantine failures [34, 81] require $3f + 1$ nodes to tolerate a total of f Byzantine and fail-stop failures. However, many practical systems do not tolerate such Byzantine faults. When nodes cannot trust each other (for example, in peer-to-peer networks such as blockchains), the system must be Byzantine fault-tolerant. However, in this dissertation, we focus on distributed storage systems deployed within a single administrative domain and thus, the servers and clients can trust each other. Therefore, the fault model we consider does not include Byzantine faults.

2.2.3 Storage Faults

Each node in a distributed storage system runs atop a local storage stack to store and manage user data. The layers in a storage stack consist of many complex hardware and software components [8, 9]. At the bottom of the stack is the media (a disk or a flash device). The firmware above the me-

dia controls the functionalities of the media. Commands to the firmware are submitted by the device driver. File systems manage these lower layers and provide interfaces to applications to use the storage. File systems can encounter faults for a variety of underlying causes including media errors, mechanical and electrical problems in the disk, bugs in firmware, and problems in the bus controller [24, 25, 97, 112, 130, 155, 156]. Sometimes, corruptions can arise due to software bugs in other parts of the operating system [39], device drivers [165], and sometimes even due to bugs in file systems themselves [56].

Due to these reasons, two problems arise for file systems: *block errors*, where certain blocks are inaccessible (also called latent sector errors) and *block corruptions*, where certain blocks do not contain the expected data.

File systems can observe block errors when the disk returns an explicit error upon detecting some problem with the block being accessed (such as in-disk ECC complaining that the block has a bit rot) [25, 155]. A previous study [25] of over 1 million disk drives over a period of 32 months has shown that 8.5% of near-line disks and about 1.9% of enterprise-class disks developed one or more latent sector errors. More recent results show similar errors arise in flash-based SSDs [97, 112, 156]. Similarly, a recent study on flash reliability [156] over a period of six years has shown that as high as 63% and 2.5% of millions of flash devices experience at least one read and write error, respectively. Recent work from Tai et al. [166] show that recent flash devices such as QLC NAND drives experience significantly higher uncorrectable bit errors than other technologies (such as SLC drives).

File systems can receive corrupted data due to a misdirected or a lost write caused by bugs in drive firmware [24, 124] or if the in-disk ECC does not detect a bit rot. Block corruptions are insidious because blocks become corrupt in a way not detectable by the disk itself. File systems, in many cases, obliviously access such corrupted blocks and silently return

them to applications. Bairavasundaram et al., in a study of 1.53 million disk drives over 41 months, showed that more than 400,000 blocks had checksum mismatches [24]. Anecdotal evidence has shown the prevalence of storage errors and corruptions [46, 73, 151]. Given the frequency of storage corruptions and errors, there is a non-negligible probability for file systems to encounter such faults.

In many cases, when the file system encounters a fault from its underlying layers, it simply passes it as-is onto the applications [130]. For example, the default Linux file system, ext4, simply returns errors or corrupted data to applications when the underlying block is not accessible or is corrupted, respectively. In a few other cases, the file system may transform the underlying fault into a different one. For example, btrfs and ZFS transform an underlying corruption into an error; when an underlying corrupted disk block is accessed, the application will receive an error instead of corrupted data [187]. In either case, we refer to these faults encountered by applications running atop the file system as *storage faults*.

Given this, it is important for applications that run on local file systems to tolerate storage faults. It is even more critical for large-scale application deployments to handle such faults because they often tend to use cheap hardware (because it is more economical than buying expensive hardware). Prior studies have shown that such cheap nearline disks are more prone to faults than enterprise-class devices [22]. Many seminal systems have been built in this spirit. For instance, Ghemawat et al. describe how they employ end-to-end checksum-based detection and recovery in the Google file system as the underlying IDE disks would often corrupt data on the chunk servers [62]. Similarly, lessons from Google [50] in building large-scale Internet services emphasize how reliability must be built into software layers running atop unreliable hardware. Given this, in this dissertation, we examine a fundamental question: do modern distributed storage systems detect and recover from storage faults correctly? Do they

use the inherent redundancy to recover from local storage faults on one or a few nodes?

2.3 Leader-based Majority Systems

In Chapters 4 and 5, we focus on leader-based majority systems. Specifically, we built CAD and cross-client monotonicity for leader-based majority systems. Thus, in this section, we provide a brief overview of such systems.

As we discussed earlier, replicas in a distributed system need to be kept consistent (identical with each other). However, this becomes complicated in the presence of concurrent requests and failures. For example, consider a simple distributed system with two replicas storing a data item a. Two clients wish to concurrently update this data item. Client c_0 wishes to set a to 0, while the other client c_1 wishes to update a to 1. If these requests are received by the replicas in different orders, then they will diverge. Specifically, the request from c_1 may arrive before c_0's request at one replica and the other way around at the other replica, leading to divergence. Similarly, failures also cause complications. For instance, after c_0 updates a on both replicas, c_1 might send its request only to one replica and fail before sending the request to the other, again causing divergence.

One common way distributed systems solve this problem is designating one of the replicas as the *leader* of the system; other nodes are referred to as *followers*. The leader is associated with an *epoch*: a slice of time; for a given epoch, at most one leader can exist [19, 120]. In this setup, clients send their requests only to the leader. The leader then establishes a single order for the requests and then replicates the requests in order to the followers. The intuition is that if all the nodes (leader and followers) start from the same initial state, apply the same sequence of commands in or-

der, then the nodes will have identical states.

In addition to ensuring order, the system must ensure that updates survive node failures. As we discussed above, when a node crashes, it loses its volatile state. Therefore, the nodes store the commands persistently on an on-disk log on their local storage medium. If a node fails, it can replay the operations from a log to recreate the volatile state it lost.

To summarize, the update protocol in a leader-based system works as follows: the client first sends a request to the leader. The leader appends the request to its on-disk log. Each log entry is uniquely identified by the epoch in which it was appended and the position of the update in the log (i.e., the log index). The leader then persists the appended request to storage and also sends the request to the followers. The followers persist the request to their logs and acknowledge the leader. Once a sufficient number of followers have acknowledged, the leader applies the request (for example, update a key-value pair), and sends the response back to the client. At this point, the client request is said to be *committed*, and thus the storage system must always recover this update, irrespective of failures.

The number of followers the leader waits for in the critical path varies across different distributed systems. For example, this could range from none to all. In this dissertation, we focus on a special class of systems, that wait for a majority of nodes to acknowledge a request (including the leader) before considering a request to be committed. Majority is defined as $\lfloor n/2 \rfloor + 1$ nodes in a system with n nodes (e.g., 3 out of 5 servers). Such systems are available as long as a majority of nodes are up and functioning; in other words, the system can tolerate the crash or partition of a minority of nodes.

The leader constantly sends heartbeats to the followers to check if they have failed. If a follower fails, the leader can continue accepting requests as long $\lfloor n/2 \rfloor$ followers are available (and following the leader). When a

failed follower recovers, it can just follow the leader; the leader sends the commands the follower missed (in order) bringing the follower up-to-date. The leader steps down if it does not hear from a majority (including self) for a while.

The system must also deal with leader failures. If the followers do not hear from the leader for a while, they become candidates and try to elect a new leader. At a high level, leader election works as follows. A candidate sends a vote request to other nodes. A node p votes for another node q if p has not already voted for another candidate and if q's log is at least as complete as the log of p. The candidate that receives votes from at least a majority (including self) is elected as the leader. This ensures that only a node that has all the committed data becomes the leader; this is a property that many practical leader-based majority systems ensure [12, 120].

2.4 Consistency Models

One of the contributions of this dissertation is a new consistency model; thus, we now explain some of the existing models. As we discussed earlier, multiple clients may submit operations to the storage system concurrently. A consistency model defines the permissible orderings of these operations submitted by clients and restricts what results can be returned by an operation [7, 176]. In this thesis, we focus on non-transactional semantics (i.e., single-object semantics); thus, we do not discuss guarantees that arise in the context of transactional distributed systems such as external consistency [45] and serializability [6].

2.4.1 Strong Consistency

Linearizability [7, 70] is a strong consistency model that hides from clients the fact that multiple copies exist. Under this consistency model, each operation appears to take effect instantaneously between its invocation and

completion. Linearizability also establishes a real-time total ordering of operations. If an operation op_2 starts after another operation op_1 completes, then op_2 must be ordered after op_1 and see the effects of op_1. For example, if op_1 updates a data item, and op_2 reads the item, op_2 must at least return the value written by op_1. Consequently, clients are never exposed to stale data in a linearizable system. If op_2 starts after op_1 starts but before op_1 completes, then these operations are concurrent. Such concurrent operations can be ordered in any way, but this order must be the same across replicas. Thus, linearizability also prevents the case where a client reads an updated state at one point, and subsequently, another client reads an older state.

Sequential consistency [84] is weaker than linearizability. It establishes a total ordering of operations across replicas; however, the order need not reflect the real-time order except for operations initiated in a single client. Therefore, in this model, clients need not see the latest data written by other clients. If a client reads some data, then the same client must at least read that data, while a different client need not see that data. However, all updates are ordered in the same order across replicas. Leader-based majority systems that establish a single order of operations at least provide sequential consistency.

2.4.2 Weaker Models

We now discuss models that do not establish total ordering and are weaker than sequential consistency and linearizability. Systems that provide these weaker guarantees are performant and provide low latencies [91]. However, they expose inconsistencies to clients and applications, providing unintuitive semantics; therefore, these models are difficult to program against [7, 169, 176].

Causal consistency. Causal consistency [23, 91] is a weaker model than

linearizability and sequential consistency. Under this model, operations are not globally ordered. Only causally related operations are ordered in the same way across clients; clients can disagree on the order of other operations. All operations from a single client (or a single thread of execution) are causally related. Two operations op_1 and op_2 are also causally related if op_1 is a write and op_2 is a read, and op_2 notices the value written by op_1. Causality is also transitive: op_1 causally precedes op_2 if op_1 causally precedes op_3 and op_3 causally precedes op_2.

Monotonic reads. Under this consistency guarantee, if a read from a client returns a value, then a subsequent read to the same data item from the same client must return the same value or a later version [169, 170]. While this guarantee may seem similar to cross-client monotonicity that we introduce in this dissertation, it is quite different. Specifically, unlike our model, this model does not ensure read monotonicity across clients or even different sessions of the same client application.

Eventual consistency. This is one of the weakest guarantees a distributed system can provide. In this model, replicas need not be identical at all times. However, when there are no updates or failures, the replicas must eventually converge and become identical [168, 169]. Under eventual consistency, there is no constraint on the ordering of operations, and reads can see any value.

2.5 Summary

In this chapter, we presented the background essential to understanding this dissertation. We described the basic concepts in a distributed storage system. We discussed the various failures possible in a distributed system, including storage faults. We provided an overview of leader-based majority systems and some of the existing consistency models.

3

Analysis of Distributed Systems Reactions to Storage Faults

In a distributed storage system, each replica works atop a commodity local file system on commodity hardware, to store and manage critical user data. However, unfortunately, as we discussed earlier (in §2.2.3), the local storage stack can sometimes return corrupted data or errors. Therefore, the ultimate responsibility of data integrity and proper error handling falls to applications that run atop local storage.

Most single-machine applications such as stand-alone databases and non-replicated key-value stores solely rely on local file systems to reliably store data; they rarely have ways to recover from local storage faults. For example, on a read, if the file system returns an error or corrupted data, these applications have no way of recovering that piece of data. Their best possible course of action is to reliably detect such faults and deliver appropriate errors to users.

Most modern distributed storage systems, much like single-machine applications, also rely upon local file systems to safely manage critical user data. However, unlike single-machine applications, distributed systems inherently store data in a replicated fashion. A carefully designed distributed storage system can potentially use this inherent redundancy to recover from errors and corruptions, irrespective of the support provided by its local file system.

Given this, in this chapter, we analyze how modern distributed storage systems react to storage faults and whether these systems preserve the durability of data in the presence of such faults. To do so, we build a fault-injection framework called CORDS that systematically injects storage faults and observes the effects of the injected faults. Using CORDS, we studied eight widely used systems: Redis [134], ZooKeeper [16], Cassandra [13], Kafka [15], RethinkDB [143], MongoDB [101], LogCabin [92], and CockroachDB [41].

A correctly designed distributed system should not be affected when data on one or a few replicas is corrupted: given that intact copies of the same data exist on other replicas, a correct system must be able to recover from such scenarios. Similarly, errors in one or a few nodes should not affect the global availability of the system given that the functionality (application code) is also replicated across many nodes.

However, from our study, we find that most distributed systems do not effectively utilize the redundancy to recover from local faults; even a single storage fault in one of the replicas can lead to catastrophic outcomes such as data loss, silent corruption, unavailability, or sometimes even the spread of corrupted data to other intact replicas. While some of these outcomes are caused due to implementation-level bugs that could be fixed by moderate developer effort, most of them arise due to a few fundamental root causes in storage-fault handling that are prevalent across many systems. This chapter is based on the paper, *Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions*, published in FAST 17 [59].

The chapter is organized as follows. First, we describe our fault model (§3.1) and how our framework injects faults and observes behaviors (§3.2). Next, we present our behavior analysis for each system (§3.3). We then derive and present a set of observations across all eight systems (§3.4). Next, we discuss features of current file systems that can impact the problems

we found (§3.5) and our experience interacting with developers (§3.6). We then discuss why systems are not tolerant of storage faults and how the problems can be potentially fixed (§3.7). Finally, we summarize and conclude (§3.8).

3.1 Fault Model

The storage devices beneath the file systems can throw errors or return corrupted data. The local file system may sometimes propagate such storage faults arising from the underlying devices *as-is* to distributed systems running atop it. In some cases, the file system may also convert some of these faults into errors. The goal of our fault model is to capture these different fault conditions that an application might encounter when running atop a local file system. While a distributed system might encounter many kinds of faults such as system crashes, network partitions, and power failures, our goal in this work is to analyze how distributed systems react to storage faults; therefore, our fault injection framework introduces only storage faults.

Our fault model has two important characteristics. First, our model considers injecting exactly a *single fault* to a *single file-system block* in a *single node* of the distributed system at a time. While correlated storage faults [24, 25] are interesting, we focus on the most basic case of injecting a single fault in a single node because our fault model intends to give maximum recovery leeway for applications. Correlated faults, on the other hand, might preclude such leeway. For example, if two or more blocks containing important application-level data structures are corrupted (possible in a correlated fault model), there might be less opportunity for the application to salvage its state. Second, our model injects faults only into application-level on-disk structures and not file-system metadata. File systems may be able to guard their own (meta)data [57];

Type of Fault		Op	Example Causes
Block Corruption	zeros	Read	lost and misdirected writes in *ext* and *XFS*
	junk	Read	lost and misdirected writes in *ext* and *XFS*
Block Error	I/O error (`EIO`)	Read	latent sector errors in all file systems, disk corruptions in *ZFS*, *btrfs*
		Write	file system mounted read-only, on-disk corruptions in *btrfs*
	Space error (`ENOSPC`, `EDQUOT`)	Write	disk full, quota exceeded in all file systems
Bit Corruption		Read	bit rots not detected by in-device ECC in *ext* and *XFS*

Table 3.1: **Possible Faults and Example Causes.** *The table shows storage faults captured by our model and example root causes that lead to a particular fault during read and write operations.*

however, if user data becomes corrupt or inaccessible, the application will either receive a corrupted block or perhaps receive an error (if the file system has checksums for user data). Thus, it is essential for applications to handle such cases.

Table 3.1 shows faults that are possible in our model during read and write operations and some examples of root causes in most commonly used file systems that can cause a particular fault. For all further discussion, we use the term block to mean a file-system block.

It is possible for applications to read a block that is corrupted (with zeros or junk) if a previous write to the underlying disk block was lost or some unrelated write was misdirected to that block. For example, in the ext family of file systems and XFS, there are no checksums for user data and so it is possible for applications to read such corrupted data, without any errors. Our model captures such cases by corrupting a block with

zeros or junk on reads.

Even on file systems such as btrfs and ZFS where user data is checksummed, detection of corruption may be possible but not recovery (unless mounted with special options such as *copies*=2 in ZFS). Although user data checksums employed by btrfs and ZFS prevent applications from accessing corrupted data, they return errors when applications access corrupted blocks. Our model captures such cases by returning similar errors on reads. Also, applications can receive `EIO` on reads when there is an underlying latent sector error associated with the data being read. This condition is possible on all commonly used file systems including ext4, XFS, ZFS, and btrfs.

Applications can receive `EIO` on writes from the file system if the underlying disk sector is not writable and the disk does not remap sectors, if the file system is mounted in read-only mode, or if the file being written is already corrupted in btrfs. On writes that require additional space (for instance, append of new blocks to a file), if the underlying disk is full or if the user's block quota is exhausted, applications can receive `ENOSPC` and `EDQUOT`, respectively, on any file system.

Our fault model also includes bit corruptions where applications read a similar-looking block with only a few bits flipped. This condition is possible when the in-disk ECC does not detect bit rots and the file system also does not detect such conditions (for example, XFS and ext) or when memory corruptions occur (e.g., corruptions introduced after checksum computation and before checksum verification [187]).

Our fault model injects faults in what we believe is a realistic manner. For example, if a block marked for corruption is written, subsequent reads of that block will see the last written data instead of corrupted data. Similarly, when a block is marked for read or write error and if the file is deleted and recreated (with a possible allocation of new data blocks), we do not return errors for subsequent reads or writes of that block. Simi-

larly, when a space error is returned, all subsequent operations that require additional space will encounter the same space error. Notice that our model does not try to emulate any particular file system. Rather, it suggests an abstract set of faults possible on commonly used file systems that applications can encounter.

3.2 Methodology

We now describe our methodology to study how distributed systems react to storage faults. We built CORDS, a fault injection framework that consists of *errfs*, a FUSE [58] file system, and *errbench*, a set of workloads and a behavior-inference script for each system.

3.2.1 System Workloads

To study how a distributed storage system reacts to local storage faults, we need to exercise its code paths that lead to interaction with its local file system. We crafted a workload suite, *errbench*, for this purpose; our suite consists of two workloads per system: read an existing data item, and insert or update a data item. Although our workloads might not exercise all code paths that lead to storage interaction, they are the most important of all workloads and give us insight as to how a particular system reacts to storage faults.

3.2.2 Fault Injection

Figure 3.1 illustrates our methodology to analyze the behavior of distributed systems. We initialize the system under study to a known state by inserting a few data items and ensuring that they are safely replicated and persisted on disk. Our workloads either read or update the items inserted as part of the initialization. Next, we configure the application

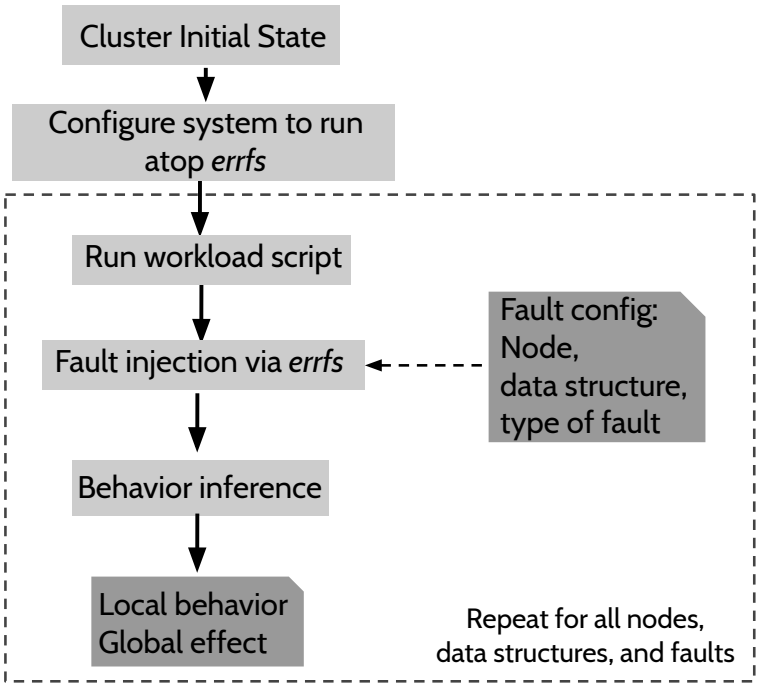


Figure 3.1: **CORDS Methodology.** *The figure shows an overview of our methodology to study how distributed systems react to local storage faults.*

to run atop *errfs* by specifying its mount point as the `data-directory` of the application. Thus, all reads and writes performed by the application flow through *errfs* which can then inject faults. We run the application workload multiple times, each time injecting a single fault for a single file-system block through *errfs*. If the application-level data structure spans multiple file-system blocks, we inject a fault only in a single file-system block constituting that data structure at a time. For bit corruptions, we flip a bit in a single field within a block at a time.

Errfs can inject two types of block corruptions: corrupted with *zeros* or *junk*. For block corruptions, *errfs* performs the read and changes the contents of the block that is marked for corruption, before returning to the application, as shown in Figure 3.2. *Errfs* can inject three types of block

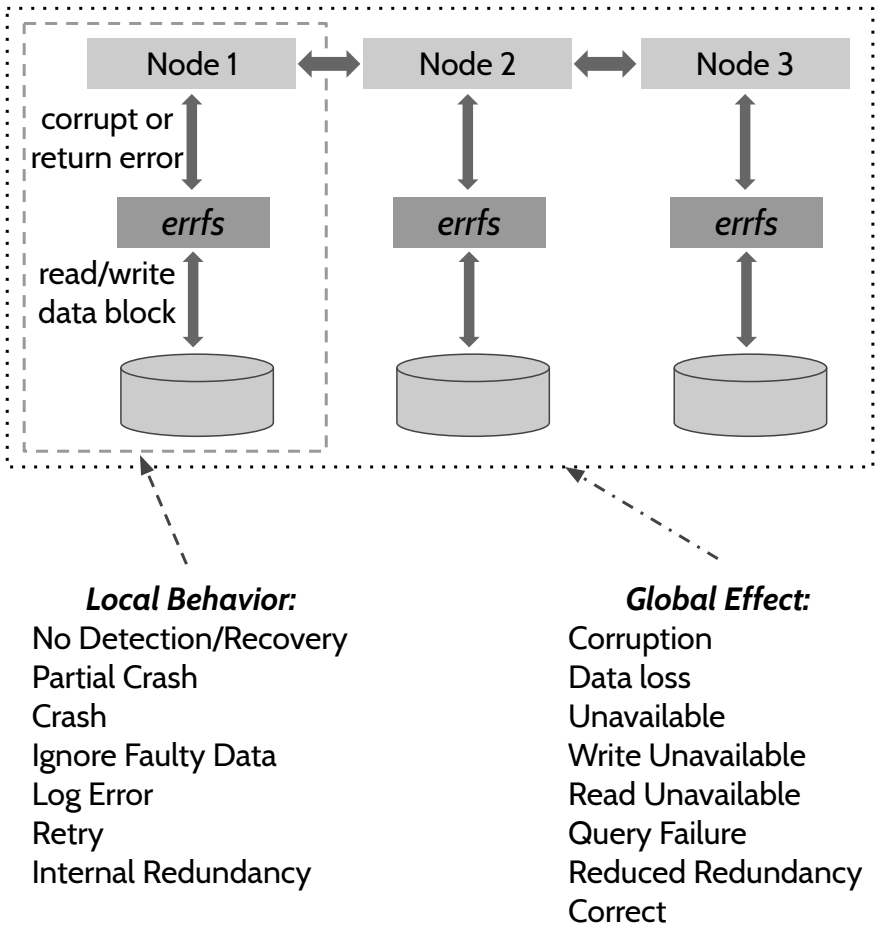


Figure 3.2: ***Errfs* and Behavior Inference.** *The figure illustrates how errfs injects faults (corruptions and errors) into a block and how we observe the local behavior and the global effect of the injected fault.*

errors: `EIO` on reads (*read errors*), `EIO` on writes (*write errors*) or `ENOSPC` and `EDQUOT` on writes that require additional space (*space errors*). To emulate errors, *errfs* does not perform the operation but simply returns an appropriate error code. For bit corruptions, *errfs* requires application-specific information consisting of various fields within a block along with their offsets and lengths. To inject a bit corruption, *errfs* flips a bit in the field that is marked for corruption before returning the data.

3.2.3 Behavior Inference

For each run of the workload where a single fault is injected, we observe how the system behaves. Our system-specific behavior-inference scripts glean system behavior from the system's log files and client-visible outputs such as server status, return codes, errors (`stderr`), and output messages (`stdout`). Once the system behavior for an injected fault is known, we compare the observed behavior against expected behaviors. The following are the expected behaviors we test for:

- *Committed data should not be lost*
- *Queries should not silently return corrupted data*
- *The cluster should be available for reads and writes*
- *Queries should not fail after retries*

We believe our expectations are reasonable since a single fault in a single node of a distributed system should ideally not result in any undesirable behavior. If we find that an observed behavior does not match expectations, we flag that particular run (a combination of the workload and the fault injected) as erroneous, analyze relevant application code, contact developers, and file bugs.

In a distributed system, multiple nodes work with their local file system to store user data. When a fault is injected in a node, we need to observe two things: local behavior of the node where the fault is injected and global effect of the fault, as shown in Figure 3.2.

Local Behavior. In most cases, a node locally reacts to an injected fault. A node can *crash* or *partially crash* (only a few threads of the process are killed) due to an injected fault. In some cases, the node can fix the problem by *retrying* any failed operation or by using *internally redundant* data (cases where the same data is redundant across files within a replica). Alternatively, the node can detect and *ignore* the corrupted data or just *log*

an error message. Finally, the node may *not even detect* or take any measure against a fault.

Global Effect. The global effect of a fault is the result that is externally visible. The global effect is determined by how distributed protocols (such as leader election, consensus, recovery, repair) react in response to the local behavior of the faulty node. For example, even though a node can locally ignore corrupted data and lose it, the global recovery protocol can potentially fix the problem, leading to a *correct* externally observable behavior. Sometimes, because of how distributed protocols react, a global *corruption*, *data loss*, *read-unavailability*, *write-unavailability*, *unavailability*, or *query failure* might be possible. When a node crashes as a local reaction, the system runs with *reduced redundancy* until manual intervention.

These local behaviors and global effects for a given workload and a fault might vary depending on the role played (leader or follower) by the node where the fault is injected. For simplicity, we uniformly use the terms *leader* and *follower* instead of primary and backup.

We note that our workload suite and model are *not complete*. First, our suite consists only of simple read and write workloads while more complex workloads may yield additional insights. Second, our model does not inject all possible storage faults; rather, it injects only a subset of faults such as corruptions, read, write, and space errors. However, even our simple workloads and fault model drive systems into corner cases, leading to interesting behaviors. Our framework can be extended to incorporate more complex faults and our workload suite can be augmented with more complex workloads; we leave this as an avenue for future work.

3.3 System Behavior Analysis

We studied eight widely used distributed storage systems: Redis (v3.0.4), ZooKeeper (v3.4.8), Cassandra (v3.7), Kafka (v0.9), RethinkDB (v2.3.4),

MongoDB (v3.2.0), LogCabin (v1.0), and CockroachDB (beta-20160714). We configured all systems to provide the highest safety guarantees possible; we enabled checksums, synchronous replication, and synchronous disk writes. We configured all systems to form a cluster of three nodes and set the replication factor at three. In this section, we present our detailed behavioral analysis for each system.

For each system, we describe the behaviors when block corruptions and block errors are injected into different on-disk structures. For each system, we also show the format of the on-disk files and the logical data structures in the system. The on-disk structure names take the form: *file_name.logical_entity*. We derive the logical entity name from our understanding of the on-disk format of the file. If a file can be contained in a single file-system block, we do not show the logical entity name. For a few systems (Redis, Cassandra, and Kafka), we perform a more detailed analysis by injecting bit corruptions in addition to block faults.

3.3.1 Redis

Redis is a popular data structure store, used as database, cache, and message broker. Redis uses asynchronous primary-backup replication by default. However, Redis can be configured to perform synchronous replication using the `WAIT` option [139]. Redis does not elect a leader automatically when the current leader fails.

On-disk Structures. Figure 3.3 shows the on-disk structures of Redis. Redis uses a simple appendonly log file (*aof*) to store the sequence of commands or operations that modify the database state. The appendonly file is not checksummed. Before recording a sequence of operations, a database identifier is logged; this identifier specifies the database to which the operations are to be applied when the appendonly file is later replayed. Periodic snapshots are taken from the *aof* to create a redis database

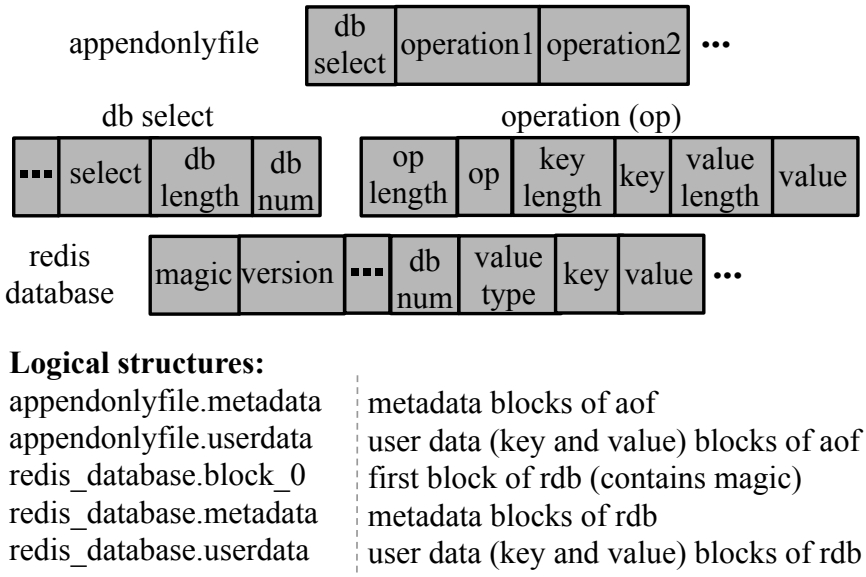


Figure 3.3: **Redis On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures of Redis. The logical structures take the following form: file_name.logical_entity. If a file can be contained in a single file-system block, we do not show the logical entity name.*

file (*rdb*). During startup, the followers re-synchronize the *rdb* file from the leader. The entire *rdb* file is protected by a single checksum.

Behavior Analysis. Figure 3.4 shows the behavior of Redis when block corruptions and block errors are introduced into different on-disk structures. When there are *corruptions* in metadata structures in the appendonly file or *errors* in accessing the same, the node simply crashes (first row of local behavior boxes for both workloads in Figure 3.4). If the leader crashes, then the cluster becomes unavailable and if the followers crash, the cluster runs with reduced redundancy (first row of global effect for both workloads).

Redis does not use checksums for user data in the appendonly file; thus, it does not detect corruptions (second row of local behavior for both

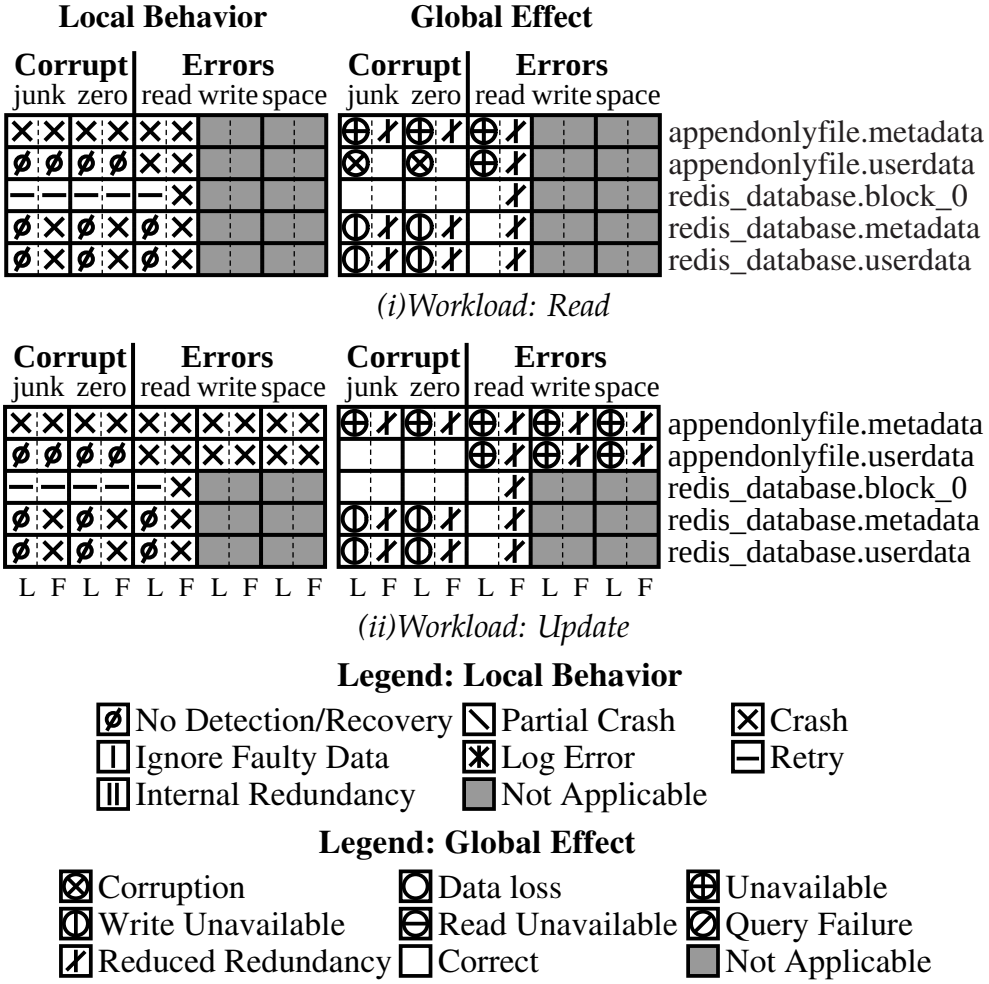


Figure 3.4: **Redis Behavior: Block Corruptions and Errors.** *The figure shows system behavior when corruptions (corrupted with either junk or zeros), read errors, write errors, and space errors are injected in various on-disk structures in Redis. Within each system workload (read and update), there are two boxes — first, local behavior of the node where the fault is injected and second, cluster-wide global effect of the injected fault. The rightmost annotation shows the on-disk logical structure in which the fault is injected. Annotations on the bottom show where a particular fault is injected (L – leader, F – follower). A gray box indicates that the fault is not applicable for that logical structure. For example, write errors are not applicable for any data structures in the read workload (since they are not written) and hence shown as gray boxes.*

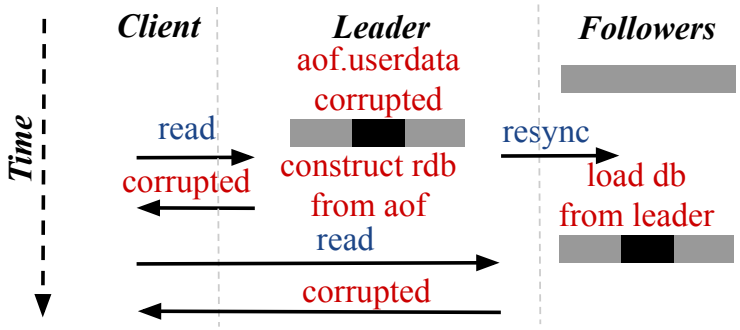


Figure 3.5: **Redis Corruption Propagation.** *The figure depicts how the re-synchronization protocol in Redis propagates corrupted user data in appendonly file (aof) from the leader to the followers, leading to a global user-visible corruption. Time flows downwards as shown on the left. The black portions denote corruption.*

workloads). If the leader is corrupted, it leads to a global user-visible corruption, and if the followers are corrupted, there is no harmful global effect (second row of global effect for read workload). Figure 3.5 shows how the re-synchronization protocol propagates corrupted user data in *aof* from the leader to the followers leading to a global user-visible corruption. The same protocol unintentionally fixes the corruption at the followers by fetching the intact data from the leader. In contrast, *errors* in appendonly file user data lead to crashes (second row of local behavior for both workloads); crashes of the leader and followers lead to cluster unavailability and reduced redundancy, respectively (second row of global effect for both workloads).

Problems in the first block of redis_database are fixed by retrying and creating the redis_database file again from data in the appendonly file (third row in Figure 3.4). When the redis_database file on a follower is corrupted, it crashes, leading to reduced redundancy. Since the leader sends the *rdb* file during re-synchronization, corruption in the same causes both the followers to crash. These crashes ultimately make the cluster unavailable for writes (fourth and fifth rows in Figure 3.4).

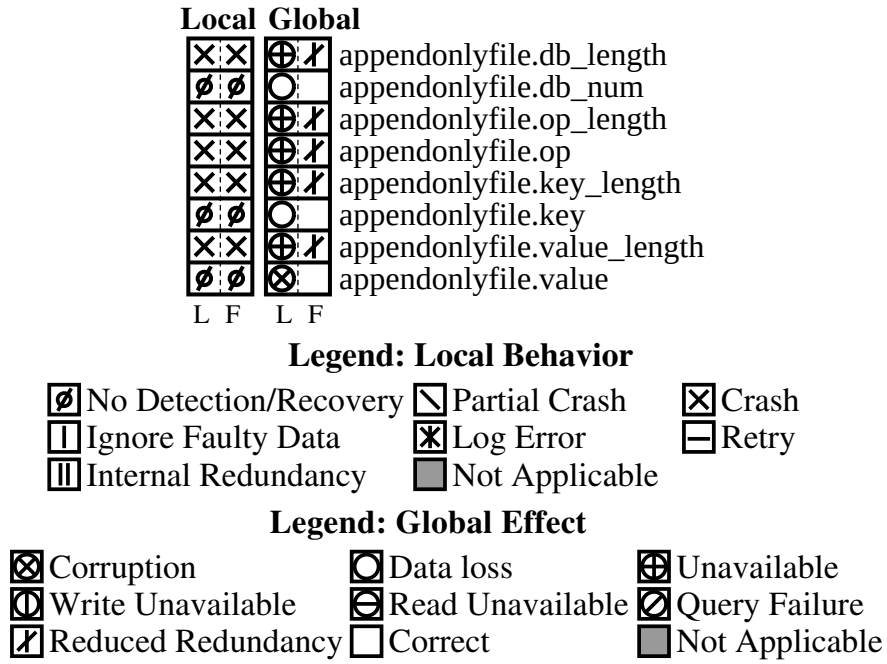


Figure 3.6: **Redis Behavior: Bit Corruptions.** *The figure shows the behavior when bit corruptions are injected. For bit corruptions, we flip a* single bit *in a field within the on-disk structure. For example, appendonlyfile.db_num is part of appendonlyfile.metadata.*

Bit Corruptions. Figure 3.6 shows the behavior of Redis when bit corruptions are injected. A single flipped bit in most of the appendonly file metadata structures results in a failed deserialization, ultimately leading to a crash. If the leader crashes, then the cluster becomes unavailable and if the followers crash, the cluster runs with reduced redundancy. On the leader, a bit flip in the key field results in a silent data loss while a bit flip in the value field results in a silent corruption.

Redis maintains a database identifier (db_num) for each database. When some data is inserted or updated, first the appropriate database (specifically, the database identifier) is recorded in the appendonly file followed by the actual update. If a bit in the recorded database identifier (P) flips

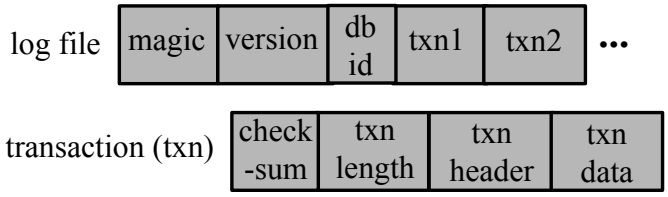
and so changes to a new value (Q), then all succeeding operations in the appendonly file are redirected to database Q instead of P. This single bit flip in the database identifier results in a silent data loss when database P is queried while supplying spurious data when database Q is queried.

In our bit-corruption experiments, we reduce the granularity of our faults: we flip a *single bit* in a field within the on-disk structure. Our bit-corruption experiments help uncover interesting behaviors not discovered through our block-corruption experiments. For instance, consider the field *appendonlyfile.db_num* which is part of *appendonlyfile.metadata*. When we inject a coarse block corruption in *appendonlyfile.metadata* on the leader (first row in Figure 3.4), the leader crashes, making the cluster unavailable. In contrast, when we inject a fine-grained bit flip in *appendonlyfile.db_num* on the leader (second row in Figure 3.6), it results in a data loss, as described above.

3.3.2 ZooKeeper

ZooKeeper is a popular service for storing configuration information, naming, and distributed synchronization. ZooKeeper provides a hierarchical name space (a *data tree*) and supports operations such as creation and deletion of nodes in the data tree. ZooKeeper implements state machine replication and uses an atomic broadcast protocol (ZAB) to maintain identical states in all the nodes in the system. The system remains available as long as a majority of the nodes are functional. It provides durability by persisting operations in a log and persisting periodic snapshots of the data tree.

On-disk Structures: Figure 3.7 shows the on-disk structures of ZooKeeper. ZooKeeper uses *log* files to append user data. The log contains a log header (magic, version, etc.) followed by a sequence of transactions. A transaction consists of a transaction header and is protected by a checksum. The transaction header contains epoch, session id, etc. ZooKeeper



Logical structures:

epoch	current and last accepted epoch
epoch_tmp	renamed to epoch after updating
myid	node id within cluster
log.transaction_head	log and transaction header
log.transaction_body	user data blocks of transaction in log
log.transaction_tail	tail blocks of transaction in log
log.remaining	unused (zero-padded) log portions
log.tail	last block of log

Figure 3.7: **ZooKeeper On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures of ZooKeeper.*

maintains two important metadata structures: *epoch* (accepted and current epoch) and *myid* (node identifier). Epochs are updated by first writing to epoch_tmp and then renaming it to epoch.

Behavior Analysis. Figure 3.8 shows the behavior when block corruptions and block errors are introduced in ZooKeeper. ZooKeeper can detect corruptions in the transaction_head and transaction_body of the *log* using checksums but reacts by simply crashing (fourth and fifth rows of local behavior for both workloads in Figure 3.8). When epoch and myid are corrupted or cannot be read, the node simply crashes (first and third rows for both workloads). Similarly, it crashes in most error cases, leading to reduced redundancy. In all crash scenarios, ZooKeeper can reliably elect a new leader, thus ensuring availability. ZooKeeper ignores a transaction locally when its tail is corrupted (sixth row of local behavior for both workloads); the leader election protocol prevents that node from

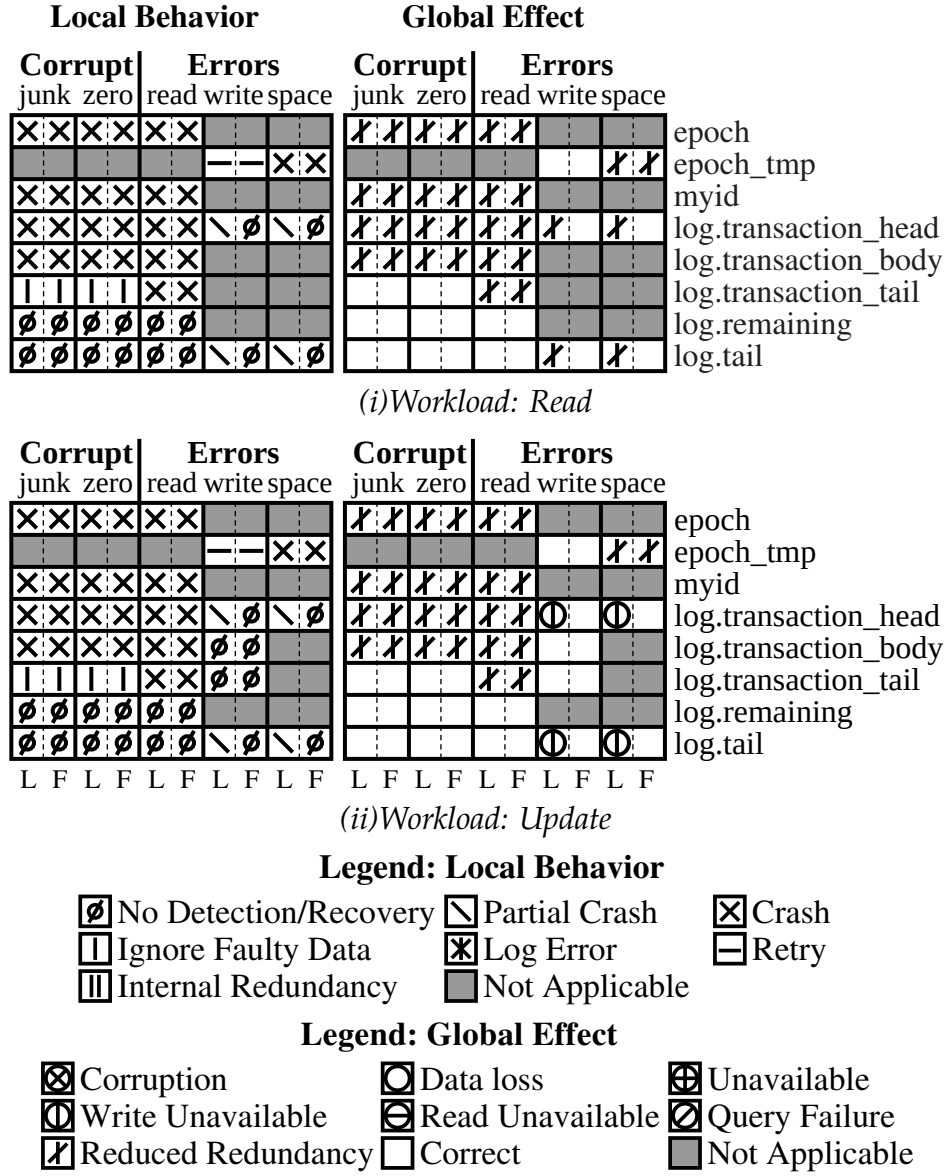


Figure 3.8: **ZooKeeper Behavior.** *The figure shows system behavior when faults are injected in various structures in ZooKeeper.*

becoming the leader. Eventually, the corrupted node repairs its log by contacting the leader, leading to correct behavior (sixth row of global ef-

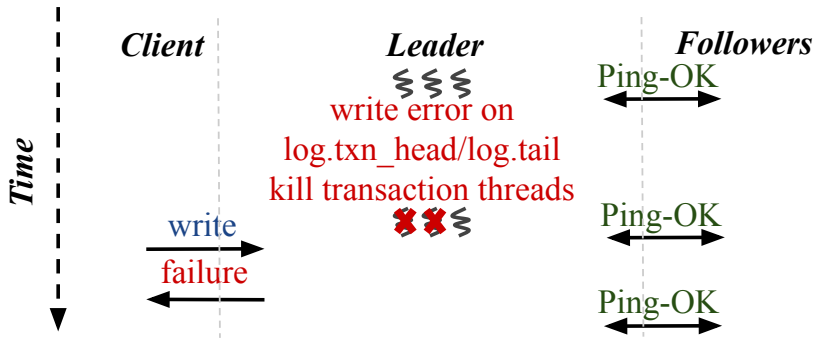
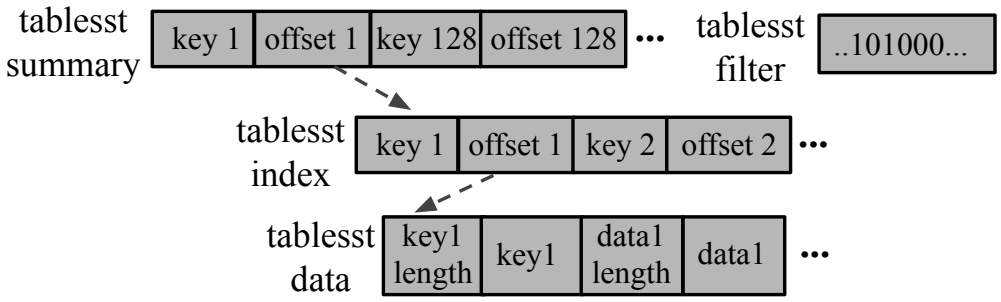


Figure 3.9: **ZooKeeper Write Unavailability.** *The figure shows how write errors lead to unavailability in ZooKeeper.*

fect for both workloads). While ignoring a transaction locally does not lead to catastrophic outcomes in our experiments, data can be lost when we introduce more than one fault (such as data corruption on multiple nodes or corruption on one node along with a lagging node), as shown by our follow-on work [10].

ZooKeeper does not recover from write errors to the transaction head and log tail (fourth and eighth rows in Figure 3.8). Figure 3.9 depicts this scenario. On write errors during log initialization, the error handling code tries to gracefully shutdown the node but kills only the transaction processing threads; the quorum thread remains alive (partial crash). Consequently, other nodes believe that the leader is healthy and do not elect a new leader. However, since the leader has partially crashed, it cannot propose any transactions, leading to an indefinite write unavailability. Notice that this scenario does not cause a harmful global effect for the read workload as reads can be locally served by any node, without requiring the leader to propose new transactions.



Logical structures:

tablesst_data.block_0	block 0 of data file in sstable for userdata
tablesst_data.metadata	meta data blocks of data file in sstable
tablesst_data.userdata	user data blocks of data file in sstable
tablesst_data.tail	metadata blocks of rdb
tablesst_index	index file in sstable
tablesst_filter	Bloom filter file in sstable
tablesst_statistics.0	block zero of statistics file in sstable
tablesst_statistics.1	block one of statistics file in sstable
tablesst_summary	summary file in sstable
tablesst_compinfo	contains compression information

Figure 3.10: **Cassandra On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures of Cassandra.*

3.3.3 Cassandra

Cassandra is a Dynamo-like [51] NoSQL store. Unlike other systems we study, Cassandra is a decentralized system; it does not have leaders and followers. The system divides all data evenly around a cluster of nodes, which form a ring. Cassandra replicates the data to a number of nodes specified by the replication factor. It also supports different read and write consistency levels. In Cassandra, rows are organized into tables and the rows are divided among nodes in the cluster based on a hash of the primary key. Cassandra also provides a SQL-like query language (CQL).

On-disk Structures: In Cassandra, the local storage engine is a variation

of log-structured merge (LSM) trees [118] that stores data in sstables. A separate sstable is maintained for each key-space; we refer to the sstables of user-created key-space as *tablesst*. Figure 3.10 shows the on-disk files in an sstable. Each sstable consists of a Bloom filter (tablesst_filter); the filter provides a fast way to determine whether a given key is present or not. If a key is found in the filter, then the table summary (tablesst_summary) and table index (tablesst_index) are accessed. The tablesst_index contains the offset of a data item in the data file (tablesst_data). The tablesst_data contains all the rows in the table.

Behavior Analysis. Cassandra enables checksum verification on user data only as a side effect of enabling compression. Therefore, we conduct two experiments in Cassandra – one with compression disabled for user tables and the other with compression enabled.

Figure 3.11(a) shows the results for block corruptions and block errors when compression is disabled for user sstables. When compression is turned *off*, corruptions are not detected on user data in tablesst_data (third row of local behavior for read workloads in Figure 3.11(a)). On a read query, a coordinator node collects and compares digests (hashes) of the data from R replicas [48]. If the digests mismatch, conflicts in the values are resolved using a *latest timestamp wins* policy. If there is a tie between timestamps, the lexically greatest value is chosen and installed on other replicas [75]. As shown in Figure 3.12, on $R = 3$, if the corrupted value is lexically greater than the original value, the corrupted value is returned to the user and the corruption is propagated to other intact replicas (third row of global effect for $R = 3$ read workload when corrupted with junk). On the other hand, if the corrupted value is lexically lesser, it fixes the corrupted node (third row of global effect for $R = 3$ read workload when corrupted with zeros). Reads to a corrupted node with $R = 1$ always return corrupted data. Faults in tablesst_index cause query failures (fifth row of global effect for read workloads). In most cases, user-visible

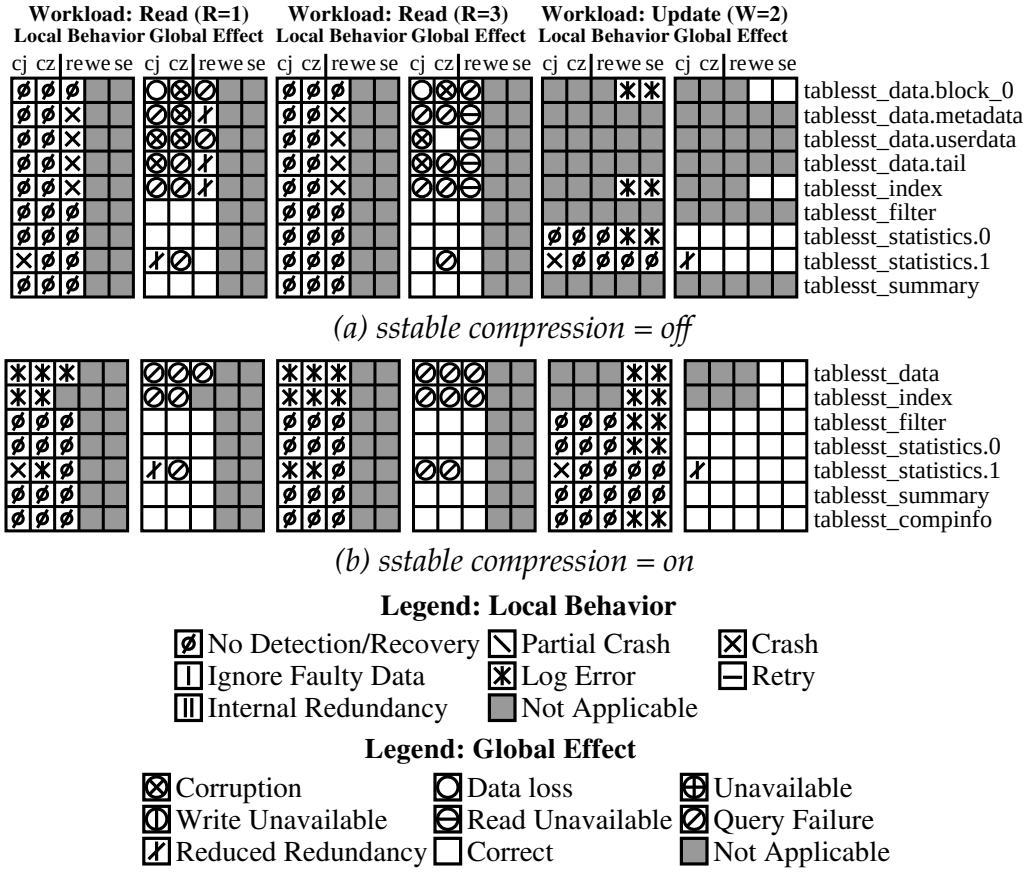


Figure 3.11: **Cassandra Behavior: Block Corruptions and Errors.** *(a) and (b) show system behavior in the presence of block corruptions (corrupted with either junk (cj) or zeros(cz)), read errors (re), write errors (we), and space errors (se) when sstable compression is turned off and turned on, respectively.*

problems that are observed in $R = 1$ configuration are not fixed even when run with $R = 3$.

Figure 3.11(b) shows the results when compression is enabled for user sstables. When compression is enabled, Cassandra maintains a checksum for every compressed block. When the value in the compressed data gets corrupted, decompression fails due to a mismatch between the stored checksum and computed checksum of the decompressed data. Thus, cor-

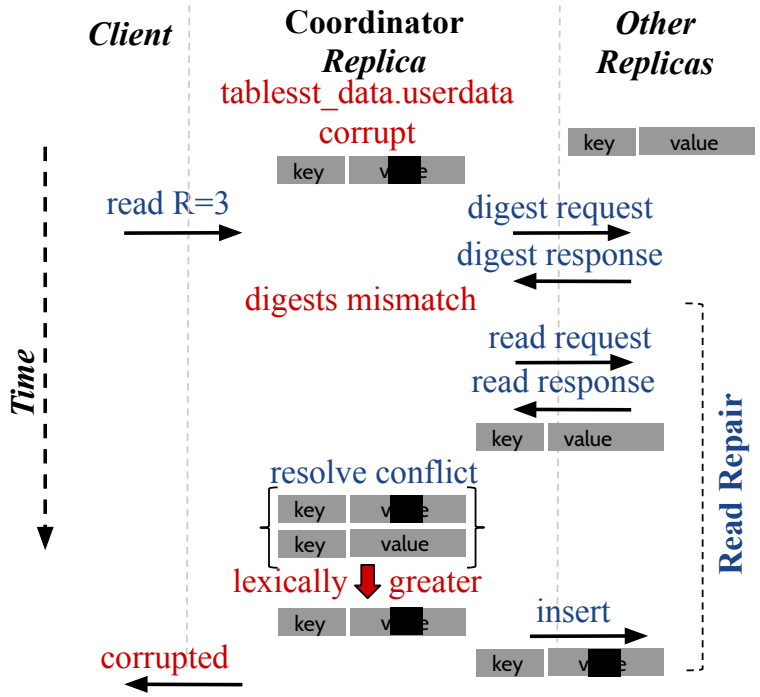


Figure 3.12: **Cassandra Corruption Propagation.** *The figure shows how the read-repair protocol in Cassandra propagates corrupted user data in a sstable file from a corrupted replica to other intact replicas.*

ruptions to tablesst_data.userdata are detected and results in failures of table scans and point queries to the data within this block (first row for read workloads in Figure 3.11(b)); point queries to the data not in this corrupted block are not affected. The corruption is not fixed automatically even when queries are run with R = 3 and results in query failures. Since we do not alter the compression feature of system schema sstables, we do not repeat this experiment for these structures.

Bit Corruptions. Figure 3.13 shows the behavior of Cassandra when bit corruptions are injected and compression is enabled for user sstables. In Cassandra, the read path involves accessing several on-disk structures. For a point query of a key, first, the key is queried in the Bloom filter (ta-

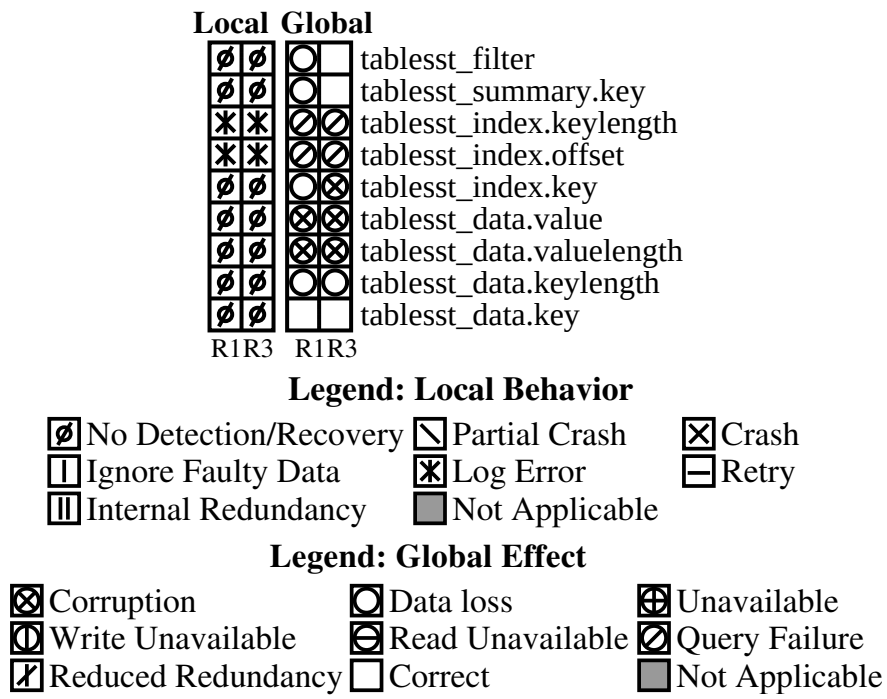


Figure 3.13: **Cassandra Behavior: Bit Corruptions.** *The figure shows the behavior in the presence of bit corruptions when sstable compression is off; the annotations on the bottom indicate the read quorum (R1 - quorum of 1, R3 - quorum of 3).*

blesst_filter); if the filter indicates the key's presence, then the table summary (tablesst_summary) and table index (tablesst_index) are accessed to determine the offset of the entry in the data file (tablesst_data). Finally, the value is read from the data file. With $R = 1$, a single bit corruption in the filter causes a data loss (first row of global effect for $R = 1$ read workload in Figure 3.13). Similarly, a single bit corruption in the key field in the table summary results in a data loss (second row of global effect for $R = 1$ read workload). With $R = 3$, the above two problems are masked (first and second rows for $R = 3$ read workload). A flipped bit in the keylength and offset of the table index results in query failures with both $R = 1$ and $R = 3$ (third and fourth rows of global effect). With $R = 1$, a corrupted

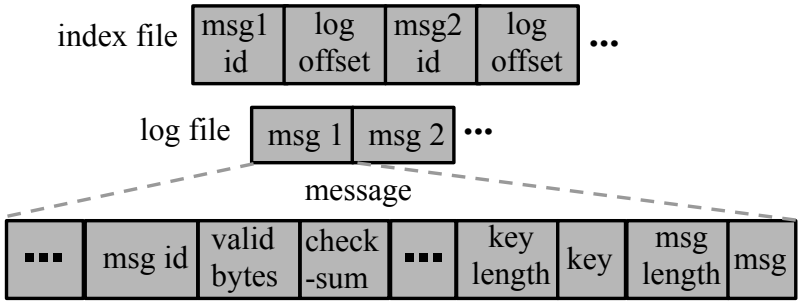
key in the table index leads to a silent data loss (fifth row of global effect). When a key K in the index is corrupted to K′, a table scan with R = 3, results in a surprising outcome: first, the scan result contains a spurious row with key K′ with the same value as the one for K; furthermore, the spurious row is propagated to all other nodes (fifth row of global effect with R = 3). A single corrupted bit in valuelength or the value in the table data results in silent corruption and corruption propagation in R = 1 and R = 3, respectively.

Introducing bit corruptions into various fields within a block helps uncover interesting behaviors in Cassandra not discovered using block-corruption experiments. For example, consider the fields *keylength*, *offset*, and *key* which are a part of *tablesst_index*. When we inject block corruptions in *tablesst_index* (fifth row in Figure 3.11(a)), it results in query failures. In contrast, when we flip a bit in the *key* of *tablesst_index*, it results in the surprising outcome where a spurious row is silently propagated to all other nodes (fifth row in Figure 3.13). Similarly, while a block corruption in *tablesst_summary* results in a correct behavior (last row in Figure 3.11(a)), a bit flip in *key* of *tablesst_summary* results in a data loss (second row for R = 1 in Figure 3.13).

3.3.4 Kafka

Kafka is a distributed persistent message queue in which clients can publish and subscribe to messages. Kafka is run as a cluster consisting of a leader and a set of followers. The system stores streams of messages in categories called topics; a topic can have zero or more consumers that subscribe to it. Each message in a topic consists of a key, a value, and a timestamp.

On-disk Structures: The on-disk structures of Kafka are shown in Figure 3.14. Incoming messages are appended to a *log* file. Each message is checksummed and is associated with a message id and an optional



Logical structures:

log.header	first block of log file
log.other	other blocks in log file
index	index file blocks
meta	file containing broker information
recovery_checkpoint	Last message flushed to disk
repl_checkpoint	Last message replicated to slaves
repl_checkpoint_tmp	Renamed to repl-ckp after updating

Figure 3.14: **Kafka On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in Kafka.*

key. Kafka maintains an *index* file which indexes messages to byte offsets within the log. Important metadata structures (such as the node identifier) are maintained in a file called *meta*. The *replication_checkpoint* and *recovery_checkpoint* structures indicate how many messages are replicated to followers so far and how many messages are flushed to disk so far, respectively. The replication offsets are updated by first writing to a temporary file (*repl_checkpoint_tmp*) and then renaming it to the final file.

Behavior Analysis. Figure 3.15 shows the behavior of Kafka when block corruptions and block errors are introduced into different structures. On read and write errors, Kafka mostly crashes. Figure 3.16 shows the scenario where Kafka can lose data and become unavailable for writes. When a log entry is corrupted on the leader, it locally ignores that entry and all subsequent entries in the log (first and second rows of local behavior

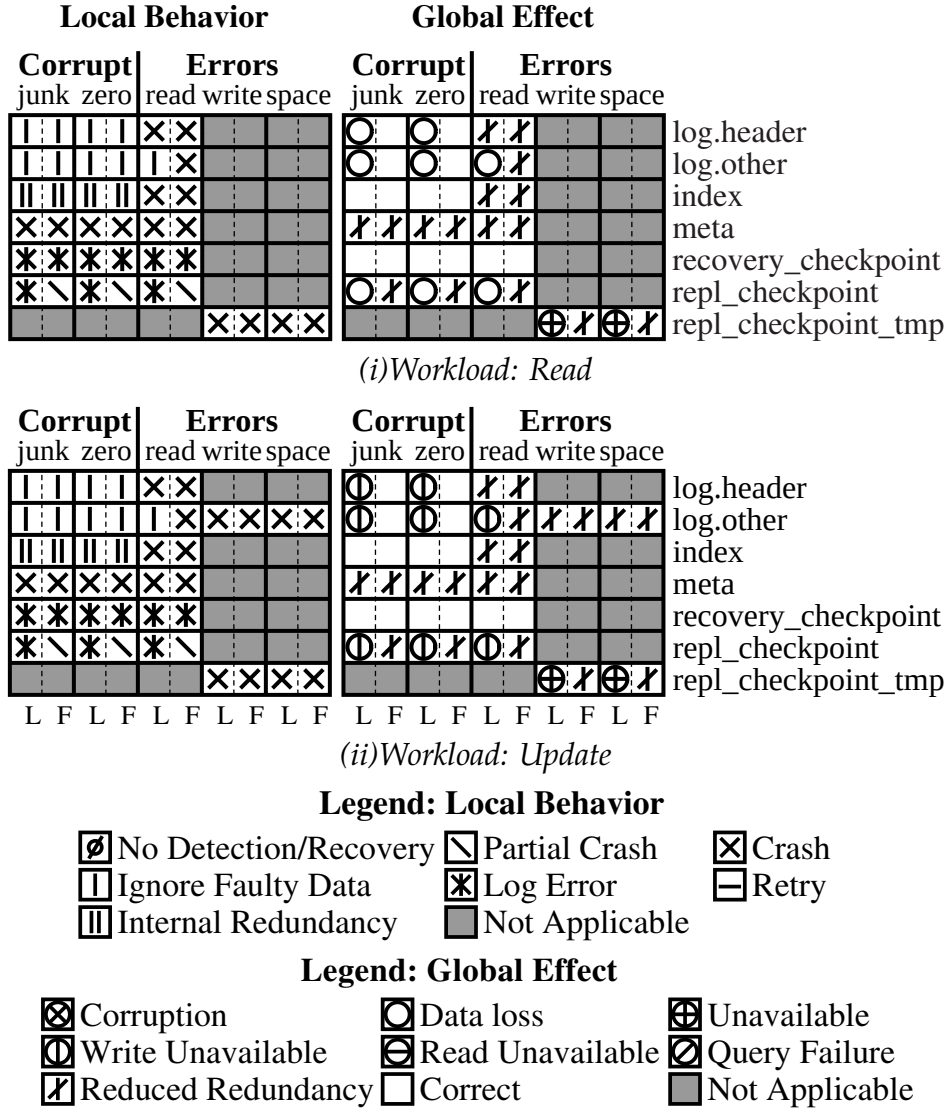


Figure 3.15: **Kafka Behavior: Block Corruptions and Errors.** *The figure shows system behavior in the presence of block corruptions and block errors.*

boxes for both workloads in Figure 3.15), resulting in a data loss. The leader then instructs the followers to do the same. On receiving this instruction from the leader, the followers check whether the leader's offset

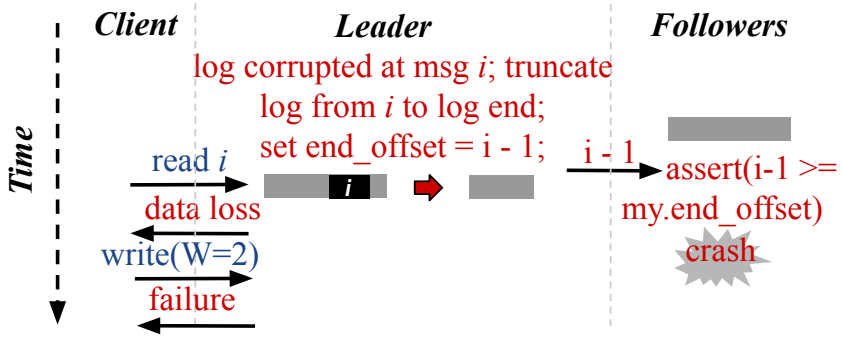


Figure 3.16: **Kafka Data Loss and Write Unavailability.** *The figure shows the scenario where Kafka loses data and becomes unavailable for writes due to a corruption.*

is greater than their checkpointed offset. If this condition does not hold, the followers hit a fatal assertion and simply crash. Once the followers crash, the cluster becomes unavailable for writes (first and second rows of global effect for write workload).

We conducted another experiment where the corruption on the leader occurs before the followers checkpoint the message offsets to their *recovery-offset-checkpoint* file. If the followers have not checkpointed the entries (that have been truncated on the leader), they truncate the entries as instructed by the leader, leading to a silent permanent data loss. In this case, the followers continue to operate without crashing.

Corruption in index is fixed using internal redundancy (third row of local behavior for both workloads). Faults in the *replication_checkpoint* of the leader result in a data loss (sixth row of global effect for read workload) as the leader is unable to record the replication offsets of the followers. Kafka becomes unavailable when the leader cannot read or write *replication_checkpoint* and *replication_checkpoint_tmp*, respectively.

Bit Corruptions. Figure 3.17 shows the behavior of Kafka when bit corruptions are injected. On a bit flip in any field of the message log, the node truncates the corrupted message and all subsequent messages. If

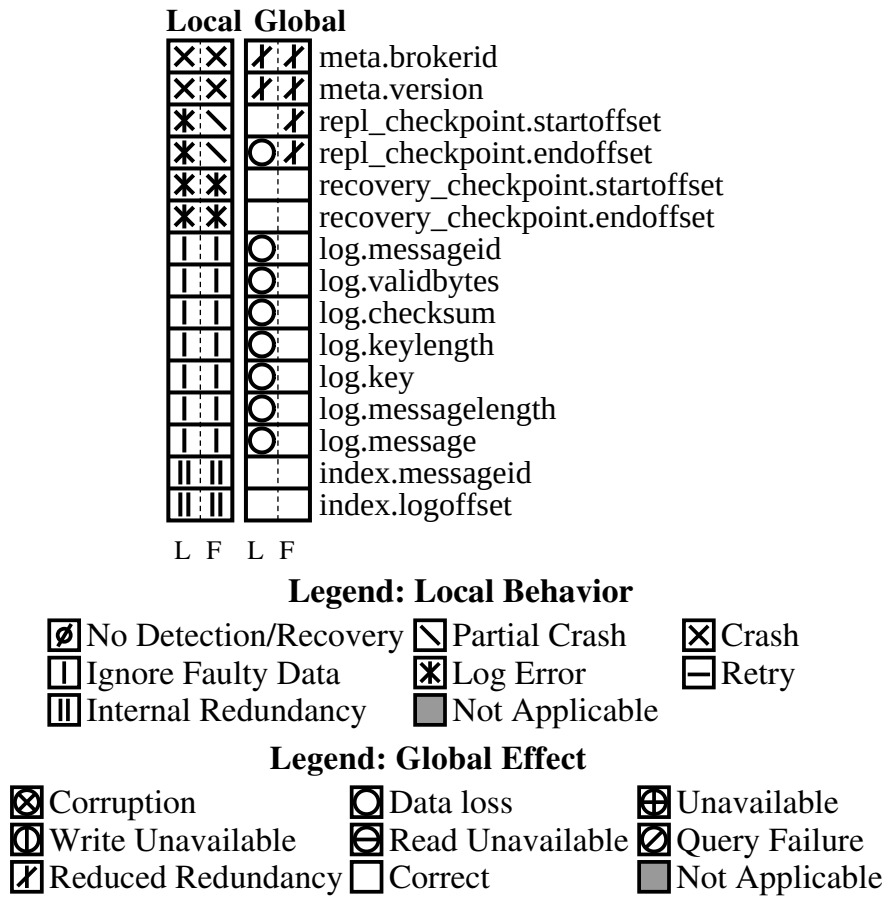


Figure 3.17: **Kafka Behavior: Bit Corruptions.** *The figure shows system behavior when bit corruptions are injected during a read workload.*

this corruption occurs on the follower, the leader supplies the truncated messages to the followers. The same single bit flip on the leader leads to a silent data loss. A bit flip in the replication offsets sometimes causes a data loss.

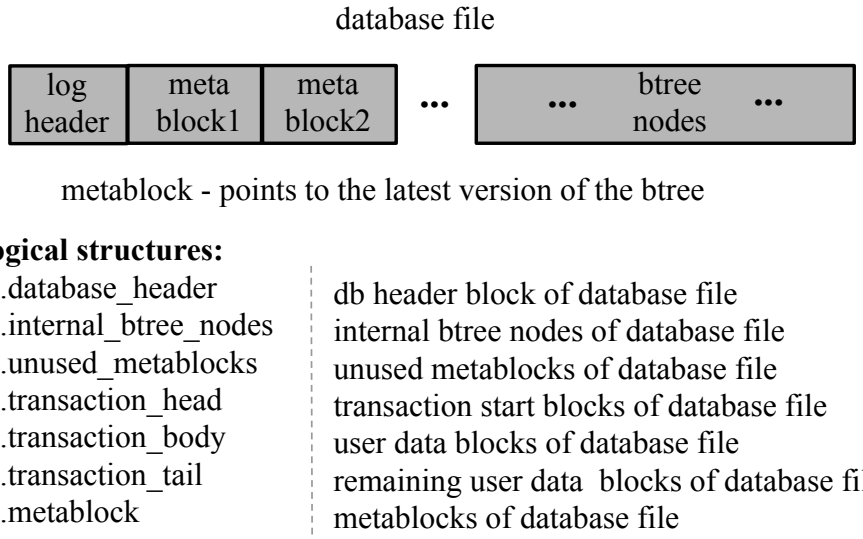


Figure 3.18: **RethinkDB On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in RethinkDB.*

3.3.5 RethinkDB

RethinkDB is a distributed database suited for pushing query results to real-time web applications [146]. RethinkDB uses the Raft consensus protocol to maintain cluster metadata. It relies on the underlying storage stack to handle data integrity and does not maintain checksums for user data.

On-disk Structures: RethinkDB uses a persistent B-tree to store all data. Transactions are stored at the leaf nodes of the tree; a transaction consists of three blocks: *db.transaction_head*, *db.transaction_body*, and *db.transaction_tail.eps*. *metablocks* in the B-tree point to the data blocks that constitute the current and the previous version of the database. On an update, new data blocks are first carefully written and flushed to disk. Then, the *metablock* with checksums is updated to point to the new data blocks, thus enabling atomic updates. The B-tree data blocks and the metablocks are part of a

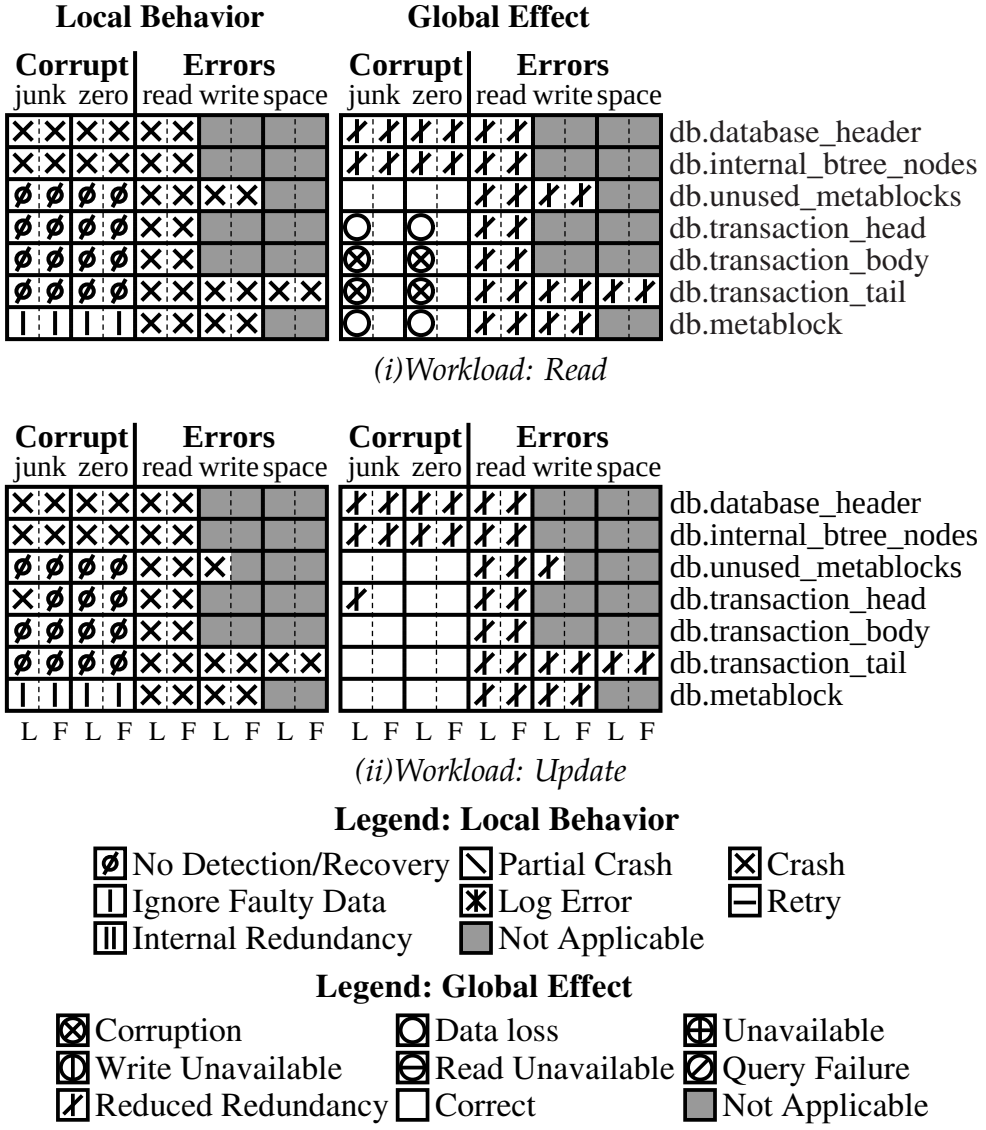


Figure 3.19: **RethinkDB Behavior.** *The figure shows system behavior when faults are injected in various on-disk logical structures.*

single database file, as shown in Figure 3.18.

Behavior Analysis. Figure 3.19 shows the behavior when block corruptions and block errors are introduced in RethinkDB. On any fault in database

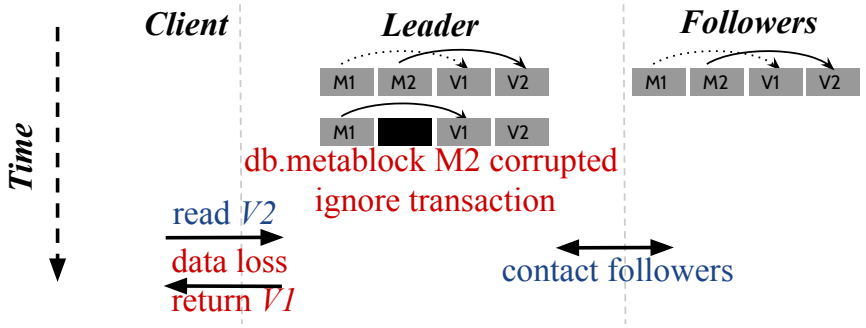
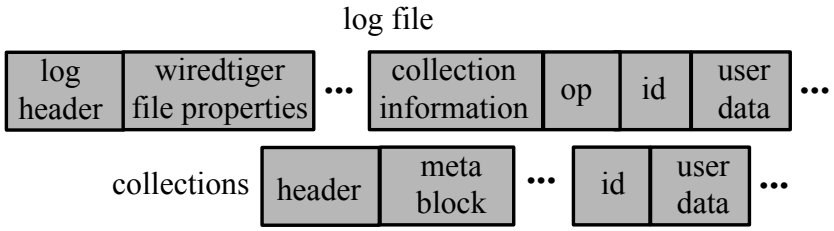


Figure 3.20: **RethinkDB Data Loss.** *The figure shows the scenario where RethinkDB exhibits data loss due to a corruption.*

header and internal B-tree nodes, RethinkDB simply crashes (first and second rows of local behavior for both workloads in Figure 3.19). If the leader crashes, a new leader is automatically elected. RethinkDB relies on the file system to ensure the integrity of data blocks; hence, it does not detect corruptions in the transaction body and tail (fifth and sixth rows of local behavior). When these blocks of the leader are corrupted, RethinkDB silently returns corrupted data (fifth and sixth rows of global effect for the leader).

Figure 3.20 depicts how data is silently lost when the transaction head or the metablock pointing to the transaction is corrupted on the leader (last row of global effect for the leader). Even though there are intact copies of the same data on the followers, the leader does not fix its corrupted or lost data, even when we perform the reads with the *majority* option. When the followers are corrupted, they are not fixed by contacting the leader. Although this does not lead to an immediate user-visible corruption or loss (because the leader's data is the one finally returned), it does so when the corrupted follower becomes the leader in the future.



Logical structures:

collections.header	header of collections file
collections.metadata	meta data blocks of collections file
collections.data	user data blocks of collections file
index	id and offsets to collections file
journal.header	header blocks of journal file
journal.other	other blocks of journal file
storage_bson	storage engine information
wiredtiger_wt	information on collections and index file

Figure 3.21: **MongoDB On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in MongoDB.*

3.3.6 MongoDB

MongoDB is a popular document-oriented database that uses JSON-like documents [101]. MongoDB provides high availability using replica sets. It uses primary-backup replication with each replica set consisting of a primary and a set of secondaries. All writes and reads are done on the primary by default. When a primary fails, the replica set automatically elects its new primary. MongoDB supports multiple storage engines. For our experiments we use WiredTiger [105] as the storage engine.

On-disk Structures: Figure 3.21 shows the different on-disk files in MongoDB. When an item is inserted or updated, it is added to the *journal* and the in-memory database is updated. If the write operation specifies the option j as true, WiredTiger forces an fsync of the journal. WiredTiger uses multi-version concurrency control, and periodically a consistent view

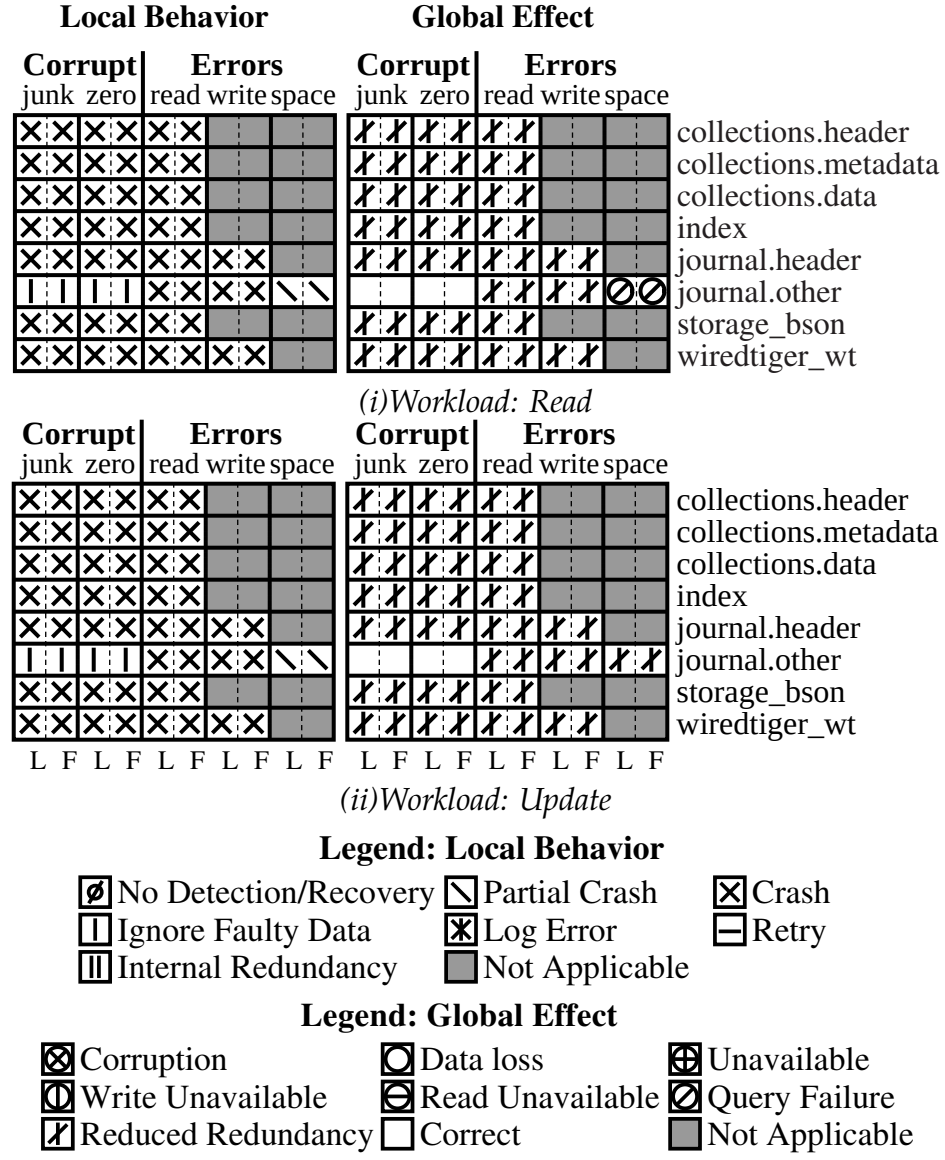


Figure 3.22: **MongoDB Behavior.** *The figure shows the system behavior when faults are injected in various on-disk logical structures in MongoDB.*

of the in-memory data is checkpointed to the *collections* file and *index* file. A master *WiredTiger* file contains information on the latest checkpoint

files. The storage engine information is stored in a *storage_bson* file.

Behavior Analysis. Figure 3.22 shows the results for block corruptions and block errors in MongoDB. MongoDB simply crashes on most errors, leading to reduced redundancy. A new leader is automatically elected if the current leader crashes. MongoDB employs checksums for all files; corruption in any block of any file causes a checksum mismatch and an eventual crash, resulting in reduced redundancy.

One exception to the above is when blocks other than journal header are corrupted. In this case, MongoDB detects and ignores the corrupted blocks (sixth row of local behavior in Figure 3.22); then, the corrupted node truncates its corrupted journal, descends to become a follower, and finally repairs its journal by contacting the leader. In a corner case where there are space errors while appending to the journal, queries fail (sixth row of global effect in Figure 3.22).

3.3.7 LogCabin

LogCabin provides a replicated and consistent data store that serves as a place for other distributed systems to maintain their core metadata such as configuration settings. LogCabin implements state machine replication and uses the Raft consensus protocol [92]. In LogCabin, all reads and writes go through the leader by default.

On-disk Structures: LogCabin implements a segmented log [153] to store data; each segment is a file on the file system. The format of a segment file is shown in Figure 3.23. There are two types of segment files—the *open* segment and the *closed* segment. The *open* segment is the current file to which data is appended. When the *open* segment is fully utilized, it is *closed* and a new segment is opened. Two metadata files (*metadata1* and *metadata2*) maintain the Raft metadata and information about the log. The metadata files are updated alternately; when a metadata file is par-

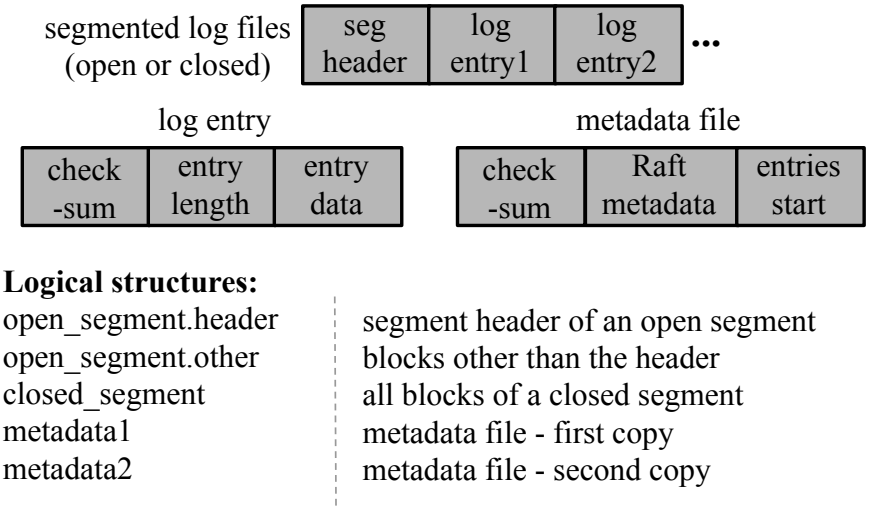


Figure 3.23: **LogCabin On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in LogCabin.*

tially updated or corrupted, LogCabin uses the other metadata file that contains slightly older metadata.

Behavior Analysis. Figure 3.24 shows the behavior when block corruptions and block errors are introduced in LogCabin. LogCabin crashes on all read, write, and space errors. Similarly, if an *open* segment file header (first row in Figure 3.24) or blocks in a *closed* segment (third row in the figure) are corrupted, LogCabin simply crashes. LogCabin recognizes corruption in any other blocks in an *open* segment using checksums, and reacts by simply discarding and ignoring the corrupted entry and all subsequent entries in that segment (second row of local behavior). If a log pointer file is corrupted, LogCabin ignores that pointer file and uses the other pointer file (fourth and fifth rows of local behavior).

In the above two scenarios, the leader election protocol ensures that the corrupted node does not become the leader; the corrupted node becomes a follower and fixes its log by contacting the new leader. This en-

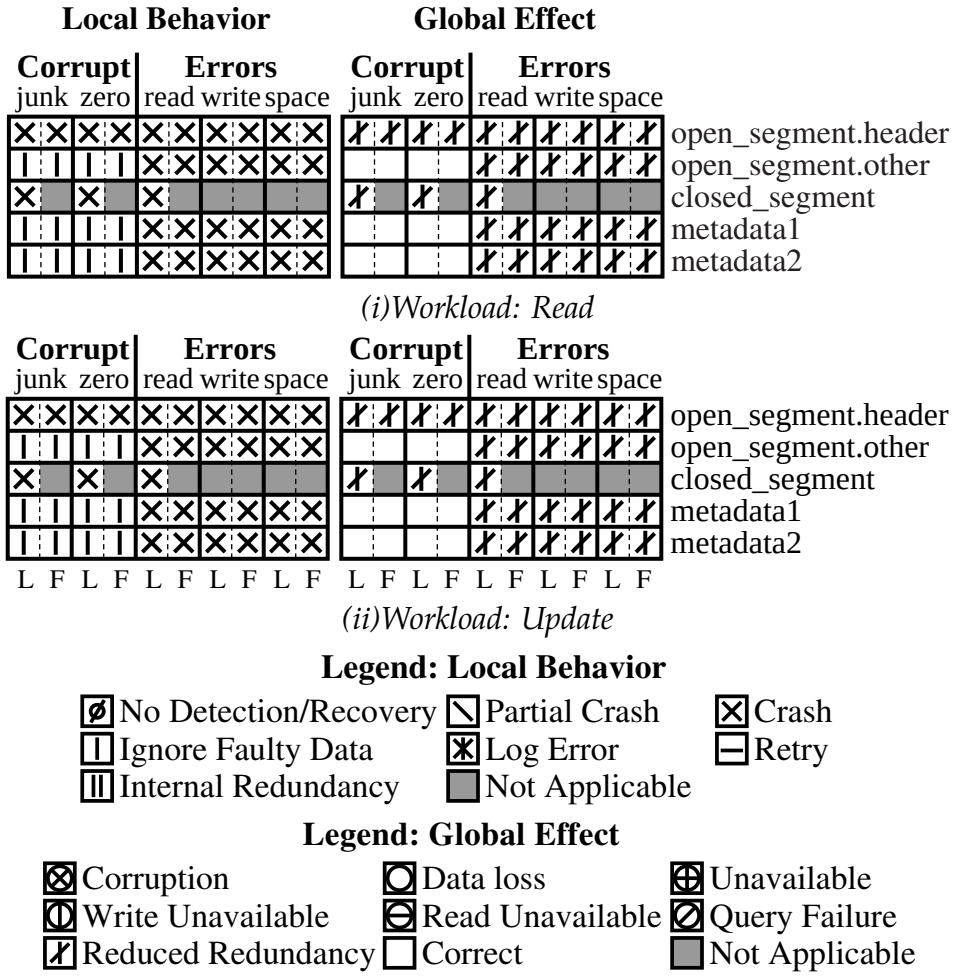
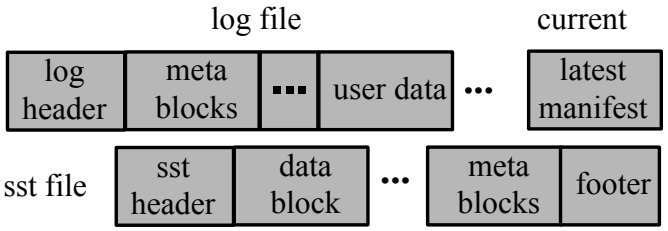


Figure 3.24: **LogCabin Behavior.** *The figure shows system behavior when faults are injected in various on-disk logical structures.*



Logical structures:

current	points to the latest manifest
manifest	information on sst files in the LSM tree
log.block_0	block 0 of log file
log.meta	meta data blocks of log file
sst.meta	meta data blocks of sst file
sst.other	remaining blocks of sst file

Figure 3.25: **CockroachDB On-disk Structures.** *The figure shows the on-disk format of the files and the logical data structures in CockroachDB.*

sures that in any fault scenario, LogCabin would not globally corrupt or lose user data. Although truncating the *open* segment does not result in a data loss in our experiments, it is possible for LogCabin to lose data in the presence of lagging nodes or data corruptions on more than one node, as shown by our follow-on work [10].

3.3.8 CockroachDB

CockroachDB is a distributed SQL database built atop a transactional and strongly-consistent key-value store. It is built to survive disk, machine, rack, and data-center failures. CockroachDB uses Raft and so as long as a majority of replicas remain available, the system can continue to make progress. It supports strongly-consistent ACID transactions and also provides an SQL-like query language [41].

On-disk Structures: Figure 3.25 shows the different on-disk files in CockroachDB. CockroachDB uses a tuned version of RocksDB for its local stor-

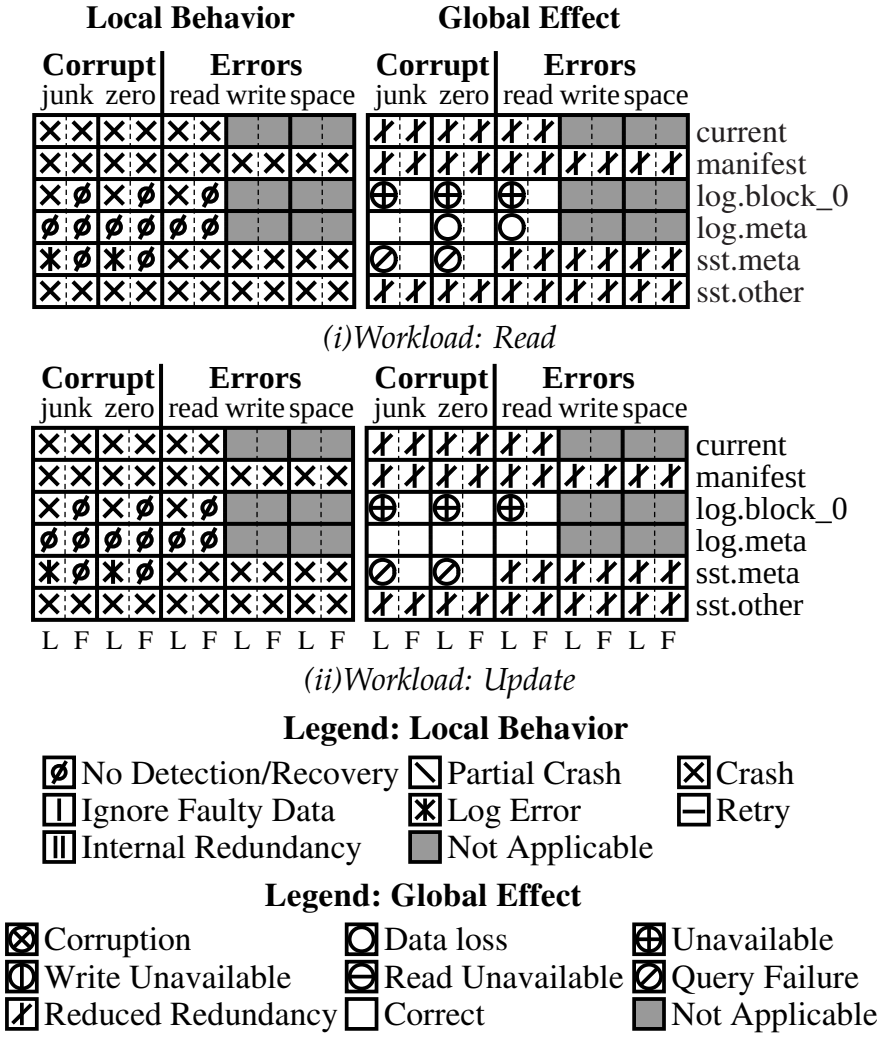


Figure 3.26: **CockroachDB Behavior.** *The figure shows the system behavior when faults are injected in various on-disk logical structures.*

age; the storage engine is an LSM tree that appends incoming data to a persistent *log*; the in-memory data is then periodically compacted to create the *sst* files. The *manifest* file lists the set of sst files that make up a particular level in the LSM tree and the *current* file points to the latest manifest.

Behavior Analysis. Figure 3.26 shows the results for block corruptions and block errors in CockroachDB. Most of the time, CockroachDB simply crashes on corruptions and errors on any data structure, resulting in reduced redundancy. Faults in the first block of the log file on the leader lead to total cluster unavailability as some followers also crash following the crash of the leader (third row of global effect). Corruptions and errors in a few other log metadata blocks can cause data loss where CockroachDB silently returns zero rows (fourth row of global effect). Corruptions in sst files cause queries to fail (fifth row of global effect) with error messages such as *table does not exist* or *db does not exist*. Overall, we found that CockroachDB has many problems in fault handling. However, the reliability may have improved in the later versions because CockroachDB was still under active development at the time of our experiments.

3.4 Observations across Systems

We now present a set of observations with respect to data integrity and error handling *across* all eight systems.

3.4.1 Systems employ diverse data integrity strategies

Table 3.2 shows different strategies employed by modern distributed storage systems to ensure data integrity. As shown, systems employ an array of techniques to detect and recover from corruption. The table also shows the diversity across systems. On one end of the spectrum, there are sys-

Technique	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Metadata Checksums	P	√	√	√	P	√	√	√
Data Checksums	P	√[a]	√[$]	√		√	√	√
Background Scrubbing								√
External Repair Tools	√		√			√		√
Snapshot Redundancy	P*	P*				P*		

P - applicable only for some on-disk structures; a - Adler32 checksum
$*$ - only for certain amount of time; $ - unused when compression is off

Table 3.2: **Data Integrity Strategies.** *The table shows techniques employed by modern systems to ensure data integrity of user-level application data.*

tems that try to protect against data corruption in the storage stack by using checksums (e.g., ZooKeeper, MongoDB, CockroachDB) while the other end of spectrum includes systems that completely trust and rely upon the lower layers in the storage stack to handle data integrity problems (e.g., RethinkDB and Redis). Despite employing numerous data integrity strategies, all systems exhibit undesired behaviors.

Sometimes, *seemingly unrelated configuration settings affect data integrity*. For example, in Cassandra, checksums are verified only as a side effect of enabling compression. Due to this behavior, corruptions are not detected or fixed when compression is turned off, leading to user-visible silent corruption.

We also find that a few systems use *inappropriate checksum algorithms*. For example, ZooKeeper uses Adler32 which is suited only for error detection after decompression and can have collisions for very short strings [95]. In our experiments, we were able to inject corruptions that caused checksum collisions, driving ZooKeeper to serve corrupted data. We believe

that it is not unreasonable to expect metadata stores like ZooKeeper to store small entities such as configuration settings reliably. In general, we believe that more care is needed to understand the robustness of possible checksum choices.

3.4.2 Faults are often undetected

We find that faults are often locally undetected. Sometimes, this leads to an immediate harmful global effect. For instance, in Redis, corruptions in the append-only file of the leader are undetected, leading to global silent corruption. Also, corruptions in the rdb of the leader are also undetected and, when sent to followers, cause them to crash, leading to unavailability. Similarly, in Cassandra, corruption of tablesst_data is undetected which leads to returning corrupted data to users and sometimes propagating it to intact replicas. Likewise, RethinkDB does not detect corruptions in the transaction head on the leader which leads to a global user-visible data loss. Similarly, corruption in the transaction body is undetected leading to global silent corruption. The same faults are undetected also on the followers; a global data loss or corruption is possible if a corrupted follower becomes the leader in future.

While some systems detect and react to faults purposefully, some react to faults only as a side effect. For instance, ZooKeeper, MongoDB, and LogCabin carefully detect and react to corruptions. On the other hand, Redis, Kafka, and RethinkDB sometimes react to a corruption only as a side effect of a failed deserialization.

3.4.3 Crashing is the most common reaction

We observe that *crashing is the most common local reaction to faults* (as is evident from the abundance of crash symbols in local behaviors of the figures in behavior analysis). Many systems do reliably detect faults (e.g.,

by using checksums for most of their on-disk data structures). However, in most cases, they simply crash on detecting a fault instead of using redundancy to recover from the fault, resulting in reduced redundancy.

Although crashing of a single node does not immediately affect cluster availability, total unavailability becomes imminent as other nodes also can fail subsequently. Also, workloads that require writing to or reading from all replicas will not succeed even if one node crashes. Moreover, since storage faults could be persistent, simply restarting does not help; the node would repeatedly crash until manual intervention fixes the underlying problem. However, such manual intervention can be often error-prone and cumbersome. We also observe that nodes are more prone to crashes on errors than corruptions.

We also observe that failed operations are rarely retried. While retries help in several cases where they are used, we observe that sometimes *indefinitely retrying operations may lead to more problems*. For instance, when ZooKeeper is unable to write new epoch information (to epoch_tmp) due to space errors, it deletes and creates a new file keeping the old file descriptor open. Since ZooKeeper blindly retries this sequence and given that space errors are sticky, the node soon runs out of descriptors and crashes, reducing availability.

Overall, although crashing may seem like a good strategy to employ, in a distributed system, there are opportunities to recover from local faults using other intact replicas.

3.4.4 Redundancy is underutilized

Contrary to the widespread expectation that redundancy in distributed systems can help recover from single faults, we observe that even a single error or corruption can cause adverse cluster-wide problems such as total unavailability, silent corruption, and loss or inaccessibility of inordinate amount of data. In many cases, almost all systems do not use redundancy

Structures	Fault Injected	Scope Affected
Redis:		
appendonlyfile.metadata	any	All[#]
appendonlyfile.userdata	read, write errors	All[#]
Cassandra:		
tablesst_data.block_0	corruptions (junk)	First Entry[$]
tablesst_index	corruptions	SSTable[#]
schemasst_compressioninfo	corruptions, read error	Table[#]
schemasst_filter	corruptions, read error	Table[#]
schemasst_statistics.0	corruptions, read error	Table[#]
Kafka:		
log.header	corruptions	Entire Log[$]
log.other	corruptions, read error	Entire Log[$*]
replication_checkpoint	corruptions, read error	All[$]
replication_checkpoint_tmp	write errors	All[#]
RethinkDB:		
db.transaction_head	corruptions	Transaction[$]
db.metablock	corruptions	Transaction[$]

[$]- data loss [#] -inaccessible [*]- starting from corrupted entry

Table 3.3: **Scope Affected.** *The table shows the scope of data (third column) that becomes lost or inaccessible when only a small portion of data (first column) is faulty.*

as a source of recovery; they miss opportunities to use other intact replicas for recovering. Notice that all the problems that we discover in our study are due to injecting only a single fault in a single node at a time. Given that the data and functionality are replicated, ideally, none of the undesirable behaviors should arise.

A few systems (MongoDB and LogCabin) automatically recover from some (not all) data corruptions by utilizing other replicas. Specifically, on encountering a corrupted entry, these systems locally ignore faulty data. Then, the leader election algorithm ensures that the node where a data item has been corrupted and hence ignored does not become the leader.

As a result, the corrupted node eventually recovers the corrupted data by fetching it from the current leader. In many situations, even these systems do not automatically recover by utilizing redundancy. For instance, LogCabin and MongoDB simply crash when closed segments or collections are corrupted, respectively.

We also find that *an inordinate amount of data can be affected when only a small portion of data is faulty*. Table 3.3 shows different scopes that are affected when a small portion of the data is faulty. The affected portions can be silently lost or become inaccessible. For example, in Redis, all user data can become inaccessible when metadata in the append-only file is faulty or when there are read and write errors in append-only file data. Similarly, in Cassandra, an entire table can become inaccessible when small portions of data are faulty. Kafka can sometimes lose an entire log or all entries starting from the corrupted entry until the end of the log. RethinkDB loses all the data updated as part of a transaction when a small portion of it is corrupted or when the metablock pointing to that transaction is corrupted.

In summary, we find that redundancy is not effectively used as a source of recovery and the general expectation that redundancy can help availability of functionality and data is not a reality.

3.4.5 Crash and corruption handling are entangled

We find that in many systems, the detection and recovery code does not try to distinguish two fundamentally distinct problems: *crashes* and *data corruption*.

Storage systems implement crash-consistent update protocols (i.e., even in the presence of crashes during an update, data should always be recoverable and should not be corrupted or lost) [22, 127, 129]. To do this, systems carefully order writes and use checksums to detect partially updated data or corruptions that can occur due to crashes.

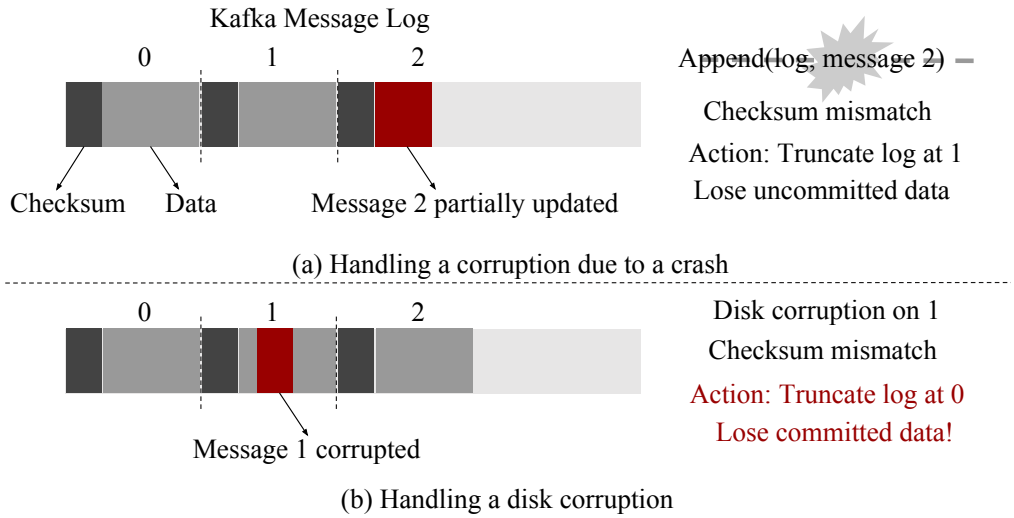


Figure 3.27: **Crash and corruption handling entanglement in Kafka.** *(a) shows how a crash during an update causes a checksum mismatch; in this case, the partially updated message is truncated. (b) shows the case where Kafka treats a disk corruption as a signal of a crash and truncates committed messages, leading to a data loss.*

A checksum mismatch for a piece of data can occur due to two distinct reasons. First, a *crash* could have occurred during the insertion of a data item and thus cause a checksum mismatch for the partially appended item. Second, a *storage corruption* can cause a checksum mismatch after the data has been successfully inserted. However, surprisingly, we find that most systems conflate these two cases: they always treat a checksum mismatch as a signal of a crash. On detecting a checksum mismatch due to corruption, all systems invariably run the crash recovery code (even if the corruption was *not* actually due to crash but rather due to a real corruption in the storage stack), ultimately leading to undesirable effects such as data loss.

We illustrate this problem by using Kafka as an example. Figure 3.27 shows how crash and corruption handling are entangled in Kafka. As shown, all incoming messages are checksummed and appended to the

message log. Figure 3.27(a) shows the case where a crash during the append of an message leaves that message partially updated, triggering a checksum mismatch during recovery. The recovery action that Kafka takes on a checksum mismatch is to truncate the message whose checksum mismatches and *all subsequent messages*. In this case, the mismatch is caused due to a partial update; hence, it is safe to truncate the message because the client has not been acknowledged of the update.

However, in contrast, consider the case shown in Figure 3.27(b); in this case, the second message has been successfully committed and the client has been acknowledged. Long after that, the disk block holding the first message gets corrupted, causing a checksum mismatch. However, the recovery code wrongly treats this corruption as a signal of a crash; hence, it truncates and loses committed messages 1 and 2. Since Kafka conflates the handling of a disk corruption and a corruption due to a crash, the node loses committed data. Similarly, ZooKeeper, on detecting a data corruption in *log.transaction_tail*, concludes that the system crashed during the last transaction commit, and truncates the log.

Another typical example of this problem is RethinkDB. RethinkDB does not use application-level checksums to handle corruption. However, it does use checksums for its metablocks to recover from *crashes*. Whenever a metablock is corrupted, RethinkDB detects the mismatch in metablock checksum and invokes its crash recovery code. The crash recovery code believes that the system crashed when the last transaction was committing. Consequently, it rolls back the committed and already-acknowledged transaction, leading to a data loss.

A few systems try to distinguish crashes from corruption. For example, LogCabin uses the following logic to do so: if a block in a closed segment (a segment that is full) is corrupted, it correctly flags that problem as a corruption and reacts by simply crashing. On the other hand, if a block in an open segment (still in use to persist transactions) is cor-

rupted, it detects it as a crash and invokes its usual crash recovery procedure. Similarly, MongoDB also differentiates corruptions in collections from journal corruptions in a similar fashion. Overall, even systems that attempt to discern crashes from corruption do not always do so correctly.

The problem of crash-corruption entanglement is not specific to distributed systems but also is applicable in any storage system that uses a log or a journal, and hence applicable to local file systems as well. We discovered later that file-system developers have encountered a bug in the ext4's journaling layer related to how ext4 handles checksum mismatches [171]. If a transaction in the journal is corrupted, ext4 replays the corrupted transaction and worse, stops processing the journal at the corrupted transaction, discarding subsequent committed transactions. The discarding of committed data is similar to the crash-corruption entanglement problem we discover. However, the best course of action that a stand-alone file system can take is to notify the user of an error; on the other hand, in a distributed system, there are multiple copies to recover from.

There is an important consequence of entanglement of detection and recovery of crashes and corruptions. During corruption (crash) recovery, some systems *fetch an inordinate amount of data to fix the problem*. For instance, when a log entry is corrupted in LogCabin and MongoDB, they can fix the corrupted log by contacting other replicas. Unfortunately, they do so by ignoring the corrupted entry and *all subsequent entries* until the end of the log and subsequently fetching all the ignored data, instead of simply fetching only the corrupted entry. Similarly, Kafka followers also fetch additional data from the leader instead of only the corrupted entry.

3.4.6 Local fault handling and global protocols interact in unsafe ways

We find that local fault-handling behaviors and commonly used distributed protocols such as leader election, read repair [51], and re-synchronization interact in unsafe ways; such unsafe interaction leads to undesirable outcomes such as propagation of corruption or data loss.

For instance, in Kafka, the local fault-handling behavior on a corrupted node interacts unsafely with the leader election protocol, turning a local data loss on the node into a global data loss. Kafka maintains a piece of metadata that contains information about replicas that are in-sync called the *in-sync-replicas* (ISR); any node in this set is guaranteed to contain all the committed data and thus is eligible to become a leader. When a log entry is corrupted on a Kafka node, it ignores the current and all subsequent entries in the log and truncates the log until the last correct entry (as we discussed earlier). Ideally, now this node should not be part of the ISR because it has lost some committed log entries. However, this node is not removed from the ISR and and is incorrectly elected the leader. Consequently, all further reads to the leader will result in a silent data loss. Kafka's synchronization protocol mandates that the followers' logs should match the leader's log. Thus, the leader also instructs the followers to truncate the log; however, this triggers an assertion at followers, resulting in their crash. As a result, all future writes become unavailable (as shown earlier in Figure 3.16). To summarize, the unsafe interaction between local behavior (i.e., to truncate the log) and the global protocol (leader election) in Kafka leads to a data loss and write unavailability.

This behavior is in contrast with the leader election protocols of ZooKeeper, MongoDB, and LogCabin where a node that has truncated log entries cannot become the leader. In these systems, a node needs to collect votes from a majority of nodes (including itself) to become the leader. A candidate will be denied the vote if it is not up-to-date as the node that is giving

the vote. In our experiments, we replicate the data items on all the nodes. Therefore, a node that has truncated log entries and lost some data cannot get votes from any other node and hence is precluded from becoming the leader. However, follow-on work [10] has shown that such systems can still lose data in the presence of corruptions. In these systems, an update is considered committed if the update reaches at least a majority of nodes. Consider a scenario where an update reaches only a bare majority; other nodes have failed, are partitioned, or are operating slowly. Now, assume one of the nodes in the bare majority encounters a corruption and truncates its data; this node can now get votes from a majority: from itself and the other nodes that have not seen the update. Once the faulty node is elected the leader, other nodes will follow the leader and truncate their logs, resulting in a global data loss.

Read-repair protocols are used in Dynamo-style quorum systems to fix any replica that has stale data. On a read request, the coordinator collects the digest of the data being read from a configured number of replicas. If all digests match, then the local data from the coordinator is simply returned. If the digests do not match, an internal conflict-resolution policy is applied, and the resolved value is installed on replicas. In Cassandra, which implements read repair, the conflict resolution resolves to the lexically greater value. When a node in Cassandra is corrupted, the corruption is not detected. If the corrupted bytes are lexically greater than the original value, the corrupted value is propagated to all other intact replicas.

Similarly, in Redis, when a data item is corrupted on the leader, it is not detected. Subsequently, the re-synchronization protocol propagates the corrupted data to the followers from the leader, overriding the correct version of data present on the followers.

To avoid these problems, an ideal system must carefully take into account how the local behaviors of a faulty node interact with the global

Catastrophic Outcomes	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Silent Corruption	×		×		×			
Unavailability	×	×	×	×				×
Data Loss	×		×	×	×			×
Query Failures			×			×		×
Reduced Redundancy	×	×	×	×	×	×	×	×

Table 3.4: **Outcomes Summary.** *The table shows the summary of our results. It shows the catastrophic outcomes caused by a single storage fault across all systems we studied. A cross mark for a system denotes that we encountered at least one instance of the outcome specified on the left.*

distributed protocols.

3.4.7 Results Summary

We now summarize our behavior analysis results. Table 3.4 summarizes the catastrophic outcomes across all distributed storage systems that we studied. The table shows that redundancy does not provide fault tolerance in many systems: a single storage fault on one node leads to undesirable outcomes such as silent user-visible corruption, unavailability, data loss, query failures, or reduced redundancy. Ideally, none of these problems should arise since we inject only a single storage fault on a single node in the system at a time.

Table 3.5 shows the fundamental root causes in storage fault handling that result in undesirable behaviors. As shown, these fundamental problems are common across all systems. First, in many systems, faults are often locally undetected. Even if faults are detected, the most common local reaction is to crash the node. All systems miss opportunities to use

Fundamental Problem	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Locally Undetected Faults	×	×	×		×			×
Crashing on Faults	×	×	×	×	×	×	×	×
Redundancy Underutilized	×	×	×	×	×	×	×	×
Crash Corruption Entangled		×		×	×	×	×	
Unsafe Protocol Interaction	×		×	×				

Table 3.5: **Observations Summary.** *The table shows the summary of fundamental problems observed across all systems. A cross mark for a system denotes that we observed at least one instance of the fundamental problem mentioned on the left.*

redundancy as a source of recovery from local storage faults. We also find that crash and corruption handling are entangled in many systems. Finally, local fault-handling behaviors and global protocols interact in unsafe ways, leading to catastrophic outcomes.

3.5 File System Implications

We now discuss features of current file systems that can impact the problems we found. All the bugs that we find can occur on XFS and all ext file systems including ext4, the default Linux file system. Given that these file systems are commonly used as local file systems in replicas of large distributed storage deployments and recommended by developers [99, 113, 131, 152], our findings have important implications for such real-world deployments.

File systems such as btrfs and ZFS employ checksums for user data; on detecting a corruption, they return an error instead of letting applications silently access corrupted data. Hence, bugs that occur due to an injected

block corruption will not manifest on these file systems. We also find that applications that use end-to-end checksums when deployed on such file systems, surprisingly, lead to poor interactions. Specifically, applications crash more often due to errors than corruptions. In the case of corruption, a few applications (e.g., LogCabin, ZooKeeper) can use checksums and redundancy to recover, leading to correct behavior; however, when the corruption is transformed into an error, these applications crash, resulting in reduced availability.

3.6 Developer Interaction

We contacted the developers of the systems regarding the behaviors we found. RethinkDB and Redis rely on the underlying storage layers to ensure data integrity [141, 142]. The RethinkDB developers intend to change the design to include application-level checksums in the future and have updated the documentation to reflect the bugs we reported [144, 145] until this is fixed. They also confirmed the entanglement between corruption and crash handling [147].

The write-unavailability bug in ZooKeeper discovered by CORDS was encountered by real-world users and has been fixed [188, 190]. The ZooKeeper developers mentioned that crashing on detecting corruption was not a conscious design decision [189]. The LogCabin developers also confirmed the entanglement between corruption and crash handling in open segments; they added that it is hard to distinguish a partial write from corruption in open segments [93]. The developers of CockroachDB and Kafka have also responded to our bug reports [42, 43, 78].

3.7 Discussion

We now discuss why modern distributed storage systems are not tolerant of single storage faults and how the problems we find can be fixed.

In a few systems (e.g., RethinkDB and Redis), we find that the primary reason is that they expect the underlying storage stack layers to reliably store data. As more deployments move to the cloud that use inexpensive storage hardware, reliable data storage might not be the reality; storage systems need to employ end-to-end integrity strategies. Lessons from building large-scale Internet services [50] also emphasize how higher layer software should provide reliability. The case for such end-to-end data integrity and error handling can also be found in the classical end-to-end arguments in system design [154].

As others have pointed out [27, 126, 127, 184], recovery code is rarely well tested, often contributing to undesirable behaviors. Similarly, although many distributed systems we study employ checksums and other resiliency techniques, recovery code that exercises such machinery is not carefully tested. We suggest that future distributed systems need to rigorously test failure recovery code using fault injection frameworks such as ours.

Third, although a body of research work [54, 155, 159, 161, 178] and enterprise storage systems [98, 122, 123] provide software guidelines to tackle partial faults, such wisdom has not filtered down to commodity distributed storage systems.

Next, while other failure models such as crash faults, network partitions, and Byzantine faults have been well studied and a vast body of work [34, 77, 81, 85, 86, 90, 117, 120] exists on how to tolerate such faults, we believe that only scant attention has been paid to problems that arise at the local storage layer in distributed storage systems. Our findings and other recent work on storage faults in distributed systems [10, 166] are initial steps in the direction to building distributed systems that tolerate

practical faults (such as storage faults) other than the traditional failure modes.

Finally, although redundancy is effectively used to provide improved availability, it remains underutilized as a source of recovery from file-system and other partial faults. To recover from storage faults, redundancy must be effectively utilized. First, the on-disk data structures have to be carefully designed so that corrupted or inaccessible parts of data can be identified. Next, corruption recovery has to be decoupled from crash recovery to fix only the corrupted or inaccessible portions of data; differentiated handling of corruptions due to crashes and other corruptions can avoid many problems. Sometimes, recovering the corrupted data might be impossible if the intact replicas are not reachable. In such cases, the outcome should be defined by design rather than left as an implementation detail. Finally, local fault-handling behavior has global implications for distributed systems. Distributed storage system developers need to understand this interaction carefully for providing improved reliability.

A few lessons from this study have proved instrumental to building new distributed systems that correctly recover from storage faults [10] (our follow-on work, not part of this thesis).

3.8 Summary and Conclusions

In this chapter, we analyzed the durability of distributed storage systems in the presence of storage faults. We presented CORDS, a tool that can systematically inject storage faults into distributed systems using which we studied eight popular systems. Our analysis revealed that tolerance of storage faults is not ingrained in modern distributed storage systems. Most systems are not equipped to effectively use redundancy across replicas to recover from local storage faults; user-visible problems such as data loss, corruption, and unavailability can manifest due to a single local stor-

age fault. Our analysis also revealed some fundamental problems in storage fault handling prevalent across many systems that lead to the above outcomes.

More broadly, even the seemingly obvious things that we take for granted in distributed systems such as *redundancy will provide fault tolerance* is not the reality in modern distributed storage systems. As these systems are emerging as the primary choice for storing critical user data, carefully designing them for all types of faults is important.

In the world of layered storage stacks that run on commodity hardware, faults are the norm, not the exception; therefore, these systems need to detect such faults carefully. Moreover, in a distributed system, several unavoidable cases such as power faults and network failures can cause nodes to be unavailable. In cases where automatic recovery is possible, simply crashing is not the optimal behavior; redundancy must be effectively utilized to recover from faults. Our testing framework, system-specific workloads, and the problems we discovered are publicly available [1].

4

Building a Stronger and Efficient Durability Primitive

In the previous chapter, we studied how even after the data has been made durable, storage faults in the local storage stack can affect the durability in distributed storage systems. In this chapter, we shift our focus and examine how a distributed system makes data durable in the first place. We refer to the set of actions a system takes to make data durable as its *durability model*. In particular, durability models describe how data is replicated and persisted across machines and the guarantees for data safety a system offers in the presence of failures. The durability model of a system strongly influences what consistency model the system can provide and how good it can perform.

However, despite this importance, this key design decision in distributed storage systems has received relatively scant attention. Therefore, in the first part of this chapter, we study existing systems on how they make data durable. We find that two durability models are popular and most systems use either one of them.

At one extreme is *immediate durability*, in which a system only returns from a write when the data has been replicated and persisted to the storage device on many nodes. At the other extreme is *eventual durability* where a system acknowledges a write after buffering it in just the memory of one or a few nodes; replication and persistence are orchestrated

in the background, eventually making the data durable. Our analysis reveals that neither of these approaches is ideal. While immediate durability enables strong consistency, it only offers poor performance; eventual durability, in contrast, delivers high performance, but it can only enable weak consistency.

To resolve this tension, in the second part of this chapter, we introduce *consistency-aware durability (*Cad*)*, a new approach to durability in a distributed storage system. The key idea behind Cad is to shift the point of durability from writes to reads. Cad delays durability upon writes, achieving high performance. Because what clients observe on reads is important for consistency models, Cad guarantees that data items are durable before serving out reads; this enables Cad to realize stronger consistency models atop it. In the next chapter, we show how *cross-client monotonic reads*, a strong consistency property can be realized upon Cad.

We implement Cad for leader-based majority systems by modifying ZooKeeper [16]. Our experiments show that ZooKeeper with Cad is significantly faster than immediately durable ZooKeeper while approximating the performance of eventually durable ZooKeeper for many workloads. This chapter is based on parts of the paper, *Strong and Efficient Consistency with Consistency-Aware Durability*, published in FAST 20 [60].

This chapter is organized as follows. We first present the different durability models and analyze how the system's durability model determines its performance and consistency guarantees (§4.1). We then describe the ideas behind the new durability model, Cad (§4.2). Next, we describe the design (§4.3) and implementation (§4.4) of Cad for leader-based majority systems. Next, we present our evaluation (§4.5). We then describe our implementation and evaluation of Cad for a different system, Redis (§4.6). We next discuss how Cad can be beneficial for current deployments, how it can be implemented in other classes of systems, and benefits of Cad even with the advent of fast storage devices (§4.7). Finally,

we summarize and conclude (§4.8).

4.1 Durability Models

A durability model describes how writes are replicated and persisted across nodes in a distributed system and what guarantees for data safety the system offers in the presence of failures. In this section, we discuss the immediate and eventual durability models and analyze how the system's durability model determines its performance and consistency guarantees. We perform this analysis for leader-based majority systems (e.g., ZooKeeper).

4.1.1 Immediate Durability

A system that employs immediate durability replicates and persists data on many nodes before clients are acknowledged. For example, in majority-based systems (such as ZooKeeper, LogCabin, and etcd [16, 53, 87, 92, 120]), upon a write, the leader synchronously replicates the data on a majority, and the nodes synchronously flush the data to disk (e.g., using *fsync*) before acknowledging clients.

By persisting data on many nodes, immediately durable systems offer strong durability guarantees: all acknowledged data can be recovered even when many or all replicas crash and recover. Given that majority-based systems remain available only when at least a majority nodes are alive, at least one node at any time will contain all committed data on its disk, and so clients will never see any data loss.

However, such strong durability comes at the cost of performance; specifically, synchronous replication and persistence incur large overheads. To highlight these overheads, we conduct a simple experiment with Redis on a five-node cluster. The replicas are located in a single data center and use SSDs for persistence. We perform a write-only workload with eight

Configuration		Throughput (ops/sec)	
Replication	Persistence	Local DC	Geo-distributed
async	async	24215	9947
sync	async	9889 (2.4× ↓)	108 (92× ↓)
sync	sync	2345 (10.3× ↓)	106 (94× ↓)

Table 4.1: **Immediate Durability Costs.** *The table shows the overheads of synchronous operations in Redis. The arrows show the throughput drop compared to the fully asynchronous configuration.*

clients where each client sends write requests to the leader in a closed loop; the leader then forwards the requests to the followers. We first configure Redis to be immediately durable where both replication and persistence are performed synchronously; at least three replicas must persist the data in the critical path of writes before acknowledgment. We enable synchronous replication and persistence in Redis by setting `WAIT` [139] to *two* and `appendfsync` [136] to *always*. We then compare the performance of this synchronous configuration with two different asynchronous configurations of Redis: synchronous replication and asynchronous persistence where data is replicated on a majority but not persisted on the disks, and asynchronous replication and asynchronous persistence where data is just buffered in the memory of the leader before acknowledgment.

Table 4.1 shows the results. As shown, when configured to replicate and persist synchronously to a majority (last row of the table), Redis is 10.3× slower than the fully asynchronous configuration (first row of the table) in which writes are buffered on one node's memory. The difference is much more pronounced if the replicas are in different data centers. To demonstrate this, we run a similar experiment but with replicas distributed across three data centers and with no data center having a majority; the clients are located in the same data center as the leader. In such a setting, immediately durable Redis is 94× slower than the fully

asynchronous configuration (last column in Table 4.1). While batching concurrent requests may improve throughput in some systems, immediate durability fundamentally suffers from high latency.

4.1.2 Eventual Durability

Given the cost of immediate durability, many deployments prefer weak, asynchronous configurations where writes are just replicated to one or a few nodes before acknowledging the client. The system then lazily replicates and persists in the background, eventually making writes durable. We refer to this durability model as eventual durability. Asynchronous operations enable the system to provide good performance; however, writes can be lost if failures arise before the system can make the data durable. However, given the performance benefits, many popular systems like Redis and MongoDB [104, 136, 137] use asynchronous configurations by *default*.

Most systems use two distinct kinds of eventual-durability configurations. In the first kind, the system performs *both* replication and persistence in an asynchronous fashion; for example, Redis, in its default configuration, buffers updates on one node's memory before acknowledgment. In the second kind, the system synchronously *replicates*, but *persists* data asynchronously; for instance, disabling the *forceSync* flag [17] in ZooKeeper achieves this effect and Redis can also be configured this way.

Asynchronous Persistence. As mentioned above, a few systems replicate synchronously but persist asynchronously: the nodes do not flush the data to disk before acknowledgment. Disabling synchronous persistence considerably boosts performance for single-data-center deployments, tempting practitioners to do so [55, 125]. For instance, as shown in Table 4.1, Redis with asynchronous persistence is 4.2$\times$ (9989/2345) faster than the synchronous version. When replicas are located in multiple data

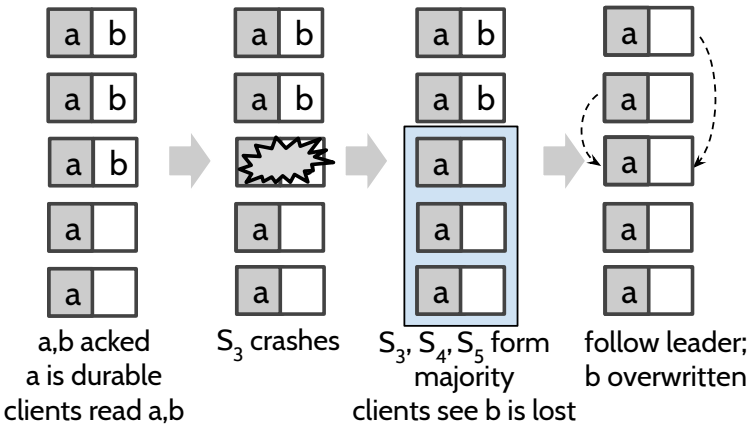


Figure 4.1: **Asynchronous Persistence.** *The figure shows how an arbitrary data loss can occur upon failures with systems that persist asynchronously. Data items shown in grey denote that they are persisted (in the background).*

centers, using asynchrony only in persistence does not offer higher performance than the synchronous configuration. This is because network round-trip latency between data centers are much higher than the time taken to persist data on SSDs. However, if the replicas run on slow HDDs, turning off persistence in the synchronous path might be beneficial even in geo-distributed settings.

With asynchronous persistence, however, the system may arbitrarily lose data although the data is replicated in memory. Surprisingly, such cases can occur although data is replicated in memory of many nodes and when just one node crashes at an inopportune moment. Consider ZooKeeper with asynchronous persistence as shown in Figure 4.1. At first, a majority of nodes (S_1, S_2, and S_3) have committed an item b, buffering it in memory; two nodes (S_4 and S_5) are operating slowly and so have not seen b. When a node in the majority (S_3) crashes and recovers, it loses b. S_3 then forms a majority with nodes that have not seen b yet and gets elected the leader. The system has thus silently lost the committed item b and so a client that previously read a state containing items a and b will now notice an older state containing only a and that b has been lost by the

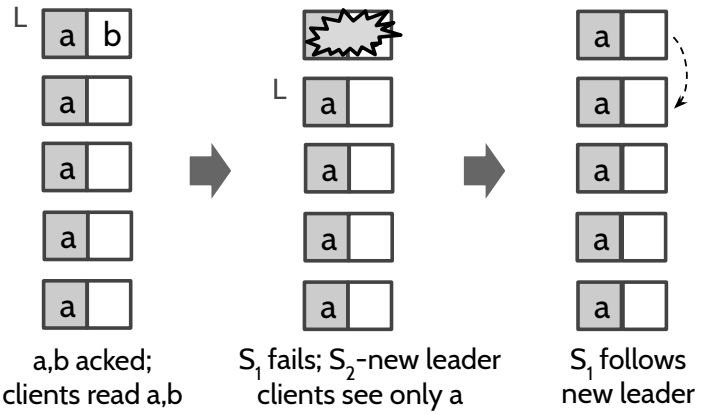


Figure 4.2: **Asynchronous Replication and Asynchronous Persistence.** *The figure shows how an arbitrary data loss can occur upon failures with systems that replicate and persist asynchronously.*

system. The intact copies on S_1 and S_2 are also replaced by the new leader because followers always follow the leader's state in many systems [120].

Data-loss instances with asynchronous persistence similar to the one shown in Figure 4.1 can be avoided if the system uses a recovery protocol like in Viewstamped Replication [90]. In such an approach, a node that has lost its data because of a crash is marked to be in a *recovering* state; such a node is precluded from participating in leader election and normal operations until it can recover its lost data by contacting a majority of nodes. By running such a recovery protocol, this approach prevents a silent data loss. However, practical systems do not employ such a strategy. Moreover, such solutions affect availability in some scenarios; for example, when a majority of nodes crash at the same time, the system will remain unavailable even after all nodes have recovered from the crash. One way to fix this problem would be to have an administrator do a repair after such failures [72]. However, such manual intervention can be error-prone; the system can *arbitrarily* lose data items that have been read by clients before the failure.

Asynchronous Replication and Asynchronous Persistence. Fully asynchronous systems, upon a write, simply buffer the data on the leader's memory, achieving high performance. However, such systems provide weak durability: they may lose data arbitrarily.

Consider the scenario shown in Figure 4.2. The leader (S_1) has acknowledged a client of item b after buffering it only in its memory. Assume, the leader fails before it can replicate the update and a few clients read the buffered item b from the leader. Once the leader fails, the other nodes (that do not have any knowledge of b) elect a new leader among themselves and the clients will now observe that the system has lost item b. The data is lost forever when the new leader overwrites the data on the old leader when it recovers.

4.1.3 Consistency and Durability

We now discuss how durability models affect consistency. Immediate durability, in addition to ensuring that the data will never be lost, acts as a foundation upon which strong consistency can be realized. For example, consider linearizability, the strongest guarantee a replicated system can provide. A linearizable system offers two properties upon reads. First, it prevents clients from seeing non-monotonic states: the system will not serve a client an updated state at one point and subsequently serve an older state to any client. Second, a read is guaranteed to see the latest update: stale data is never exposed. However, to provide such strong guarantees on reads, a linearizable system must pay the cost of immediate durability during write operations [87]. For example, majority-based linearizable systems synchronously persist data on a majority before acknowledgment [120]. Upon a synchronously durable foundation, these systems use additional mechanisms to ensure linearizability [87]. For example, in addition to using immediate durability, many practical linearizable systems restrict reads to the leader [80, 92, 108, 119].

Given that immediate durability is expensive, many systems adopt eventual durability; however, by doing so, these systems settle for weaker consistency. As we discussed earlier, eventual durability can lead to arbitrary loss of data. Upon such a weakly durable substrate, it is hard (if not impossible) to realize strong consistency. For example, consider the above two properties of strong consistency: clients can never see stale or out-of-order data. A weakly durable system violates these two properties. Data-loss instances as shown in Figures 4.1 and 4.1 naturally expose stale and out-of-order states: clients may read a state that contains items a and b before the failure and later notice the state contains only a.

In essence, systems built upon eventual durability cannot realize strong consistency properties in the presence of failures. Such systems can serve a newer state before the failure but an older one after recovery, exposing non-monotonic reads.

Only models weaker than linearizability such as causal consistency can be built atop eventual durability; such models offer monotonic reads only in the absence of failures and within a single client session. If the server to which the client is connected crashes and recovers, the client has to establish a new session in which it may see a state older than what it saw in its previous session [96].

4.2 Consistency-aware Durability: A New Durability Primitive

To summarize our discussion thus far, immediate durability enables strong consistency but is prohibitively expensive. Eventual durability offers high performance, but only weak consistency can be built upon it. Given this, we ask the following question: *is it possible to rethink the durability layer to achieve both strong consistency and high performance?*

We first note that to provide high performance the system cannot employ synchronous writes because replicating and persisting on many nodes in the critical path of writes is simply too slow. Second, we realize that what clients observe upon reads is important for most consistency models. Based on these two insights, we rethink the durability layer and propose *consistency-aware durability* or CAD, a new durability primitive.

The main idea underlying CAD is *read-triggered durability*: i.e., durability is guaranteed upon reads not writes. Similar to eventual durability, CAD allows writes to be completed asynchronously. The writes are just buffered in the memory of one node and immediately acknowledged; replication and persistence happen in the background. However, unlike eventual durability, CAD enforces durability upon reads: if a non-durable data item is read, CAD makes the item durable by synchronously replicating and persisting the item before serving the read. This ensures that data that has been read by a client is not lost and thus prevents out-of-order states.

CAD does not always incur the cost of synchronous operations when data is read. First, for many workloads, CAD can make the data durable in the background well before applications read it. Further, only the first read to non-durable data triggers synchronous replication and persistence; subsequent reads are fast. Thus, in the common case, when clients do not read data immediately after writing (which is natural for many workloads), CAD can realize the high performance of eventual durability. In the case where clients do read data immediately after writing, CAD incurs overheads but ensures durability of data that has been read by clients.

In summary, by delaying the durability of writes, CAD achieves high performance. However, by ensuring that the data is durable before it is read, CAD enables building of stronger consistency models upon it. In the next chapter, we show how one such stronger consistency property

that we call *cross-client monotonic reads* can be realized atop CAD and also discuss the utility of the new consistency guarantee.

An important aspect of CAD is that it does not offer complete freedom from data loss. CAD may lose updates if failures arise before the updates are read. Therefore, CAD is not a replacement for applications that require immediate durability. However, many systems do not use immediate durability and currently adopt eventual durability, thereby settling for weaker guarantees. CAD offers a way for these systems to realize better guarantees without forgoing performance.

4.3 CAD Design

We now describe how we design consistency-aware durability for leader-based majority systems. We first provide a brief overview of leader-based systems (§4.3.1) and outline CAD's failure model and guarantees (§5.3.1). We then describe the update path (§4.3.3) and CAD's state durability guarantee (§4.3.4). We then describe CAD's mechanisms to ensure durability on reads (§4.3.5 and §4.3.6). We finally discuss CAD's correctness (§4.3.7).

4.3.1 Leader-based Majority Systems

As discussed earlier (in §2.3), in leader-based systems (such as ZooKeeper), all updates flow through the leader which establishes a single order of updates by storing them in a log and then replicating them to the followers [71, 120]. The leader is associated with an epoch: a slice of time, in which at most one leader can exist [19, 120]. Each update is uniquely identified by the epoch in which it was appended and its position in the log. The leader constantly sends heartbeats to the followers; if the followers do not hear from the leader for a while, they elect a new leader.

With immediate durability, the leader acknowledges an update only after a majority of replicas (i.e., $\lfloor n/2 \rfloor + 1$ nodes in a n-node system) have

persisted the update. With eventual durability, updates are either buffered in memory on just the leader (asynchronous replication and persistence) or a majority of nodes (asynchronous persistence) before acknowledgment.

4.3.2 Failure Model and Guarantees

Similar to many majority-based systems, CAD intends to tolerate only fail-recover failures, not Byzantine failures [86]. In the fail-recover model, nodes may fail at any time and recover at a later point. Nodes fail in two ways; first, they could crash (e.g., due to power failures); second, they may get partitioned due to network failures. When a node recovers from a crash, it loses its volatile state and is left only with its on-disk state. During partitions, a node's volatile state remains intact, but it may not have seen data that the other nodes have.

Guarantees. CAD preserves the properties of a leader-based system that uses eventual durability; it does *not* prevent absolute data loss. For example, if failures arise after writing the data but before reading it, CAD may lose a few recent updates and thus subsequent reads can get an older state. However, CAD guarantees that data that has been read by applications can be recovered under all failure scenarios (e.g., even if all replicas crash and recover). Further, CAD maintains a single order of updates similar to leader-based majority systems. Therefore, CAD guarantees that all updates upto the latest item that was read remain durable and are not lost upon failures. Majority-based systems remain available as long as a majority of nodes are functional [20, 120]; CAD ensures the same level of availability.

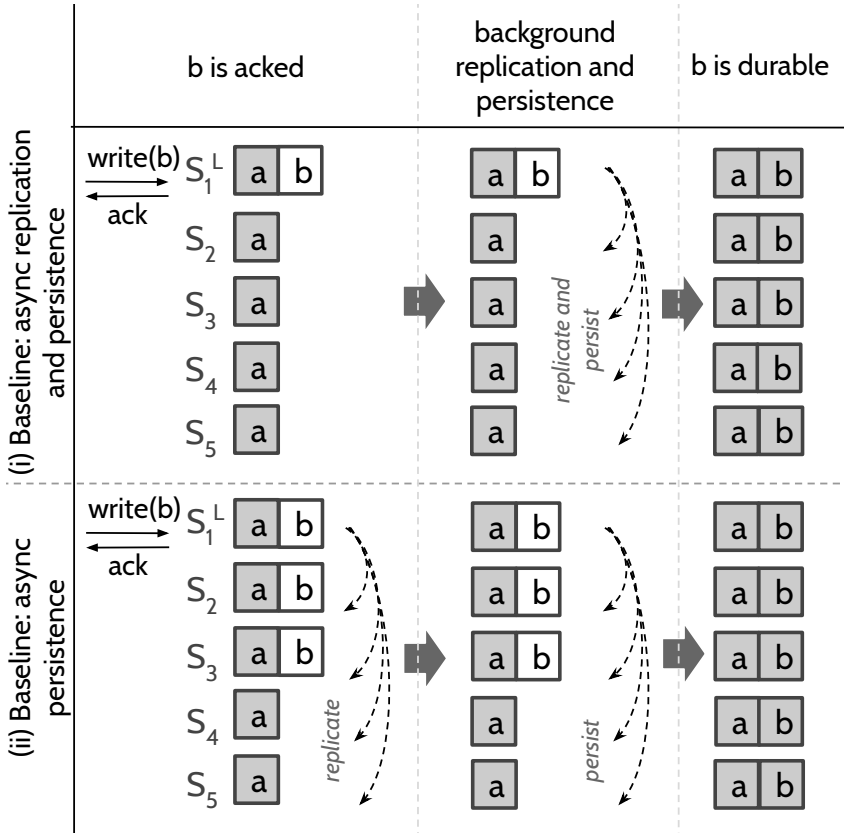


Figure 4.3: **Cad Update Path.** *The figure shows the update path in* Cad. *Data items that are durable are shown in grey boxes. In (i), the baseline performs both replication and persistence asynchronously; in (ii), the baseline synchronously replicates but persists lazily in the background. When a client writes item* b*, the write is acknowledged before* b *is made durable similar to the baselines.* Cad *then makes* b *durable in the background by replicating and persisting* b *on other nodes asynchronously.*

4.3.3 Update Path

In the rest of this section, we use eventual durability as the baseline to highlight how Cad is different from it. Cad aims to perform similarly to this baseline but provides better guarantees. We now provide intuition about how Cad works and explain its mechanisms.

Cad preserves the update path of the baseline eventual system as it

aims to provide the same performance during writes. Thus, if the baseline employs asynchronous replication and persistence, then CAD also performs both replication and persistence asynchronously, buffering the data in the memory of the leader as shown in Figure 4.3(i). Similarly, if the baseline synchronously replicates but asynchronously persists, then CAD also does the same upon writes as shown in Figure 4.3(ii). While preserving the update path, in CAD, the leader keeps replicating updates in the background and the nodes flush to disk periodically.

4.3.4 State Durability Guarantee

When a read for an item i is served, CAD guarantees that the *entire state* (i.e., writes even to other items) up to the last update that modifies i are durable. For example, consider a log such as $[a, b_1, c, b_2, d]$; each entry denotes a (non-durable) update to an item, and the subscript shows how many updates are done to a particular item. When item b is read, CAD guarantees that all updates at least up to b_2 are made durable before serving b. CAD makes the entire state durable instead of just the item because it aims to preserve the update order established by the leader (as done by the base system).

CAD considers the state to be durable when it can recover the data after any failures including cases where all replicas crash and recover and in all successive views of the cluster. Majority-based systems require at least a majority of nodes to form a new view (i.e., elect a leader) and provide service to clients. Thus, if CAD safely persists data on at least a majority of nodes, then at least one node in any majority even after failures will have all the data that has been made durable (i.e., that was read by the clients) and thus will survive into the new view. Therefore, CAD considers data to be durable when it is persisted on the disks of at least a majority of nodes.

4.3.5 Handling Reads: Durability Check

We now discuss how CAD handles reads; we use Figure 4.4 to do so. When a read request for an item i arrives at a node, the node can immediately serve i from its memory if all updates to i are already durable (e.g., Figure 4.4, read of item *a*); otherwise, the node must take additional steps to make the data durable. As a result, the node first needs to be able to determine if all updates to i have been made durable or not.

A naive way to perform this check would be to maintain for each item how many nodes have persisted the item; if at least a majority of nodes have persisted an item, then the system can serve it. A shortcoming of this approach is that the followers must inform the leader the set of items they have persisted in each response, and the leader must update the counts for all items in the set on every acknowledgment.

CAD simplifies this procedure by exploiting the ordering of updates established by the leader. Such ordering is an attribute common to many majority-based systems; for example, the ZooKeeper leader stamps each update with a monotonically increasing epoch-counter pair before appending it to the log [18]. In CAD, with every response, the followers send the leader only a single index called the *persisted-index* which is the epoch-counter of the last update they have written to disk. The leader also maintains only a single index called the *durable-index* which is the index up to which at least a majority of nodes have persisted; the leader calculates the durable-index by finding the highest persisted-index among at least a majority (including self).

When a read for an item i arrives at the leader, it compares the *update-index* of i (the epoch-counter of the latest update that modifies i) against the system's durable-index. If the durable-index is greater* than the update-index, then all updates to i are already durable and so the leader serves i

*An index a is greater than index b if (a.epoch $>$ b.epoch) or (a.epoch $==$ b.epoch and a.counter $>$ b.counter).

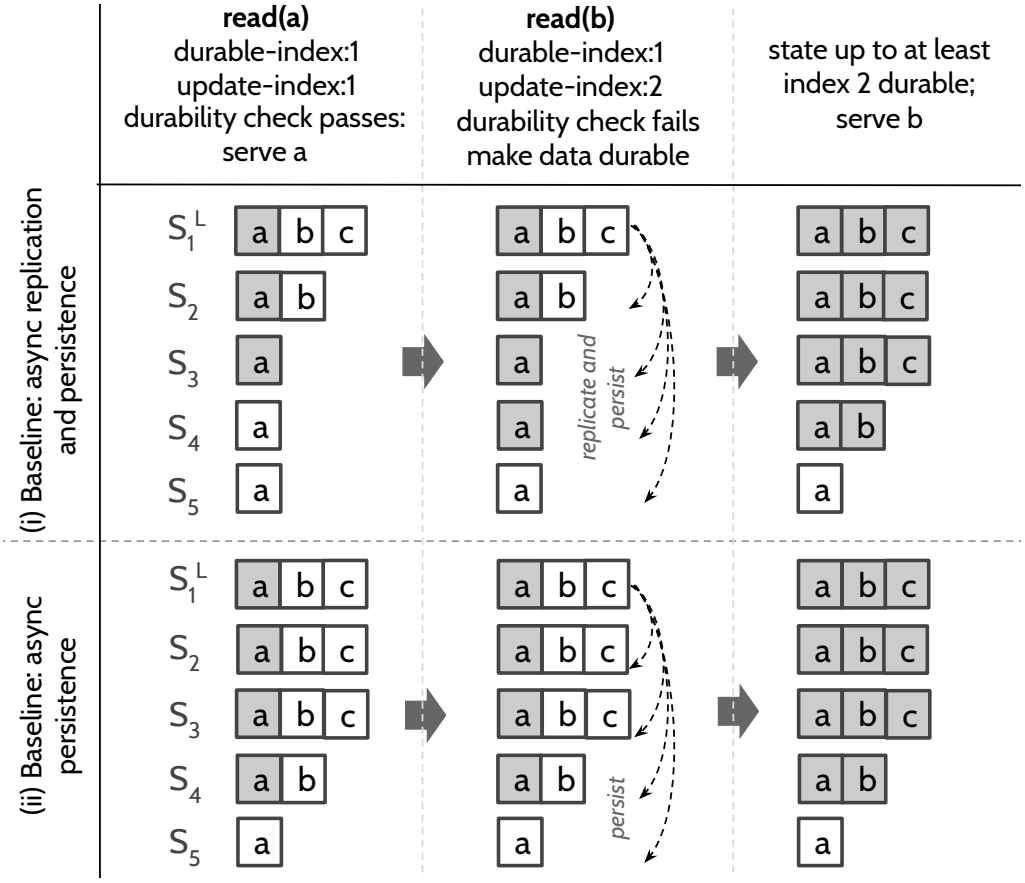


Figure 4.4: **Cad Durability Check.** *The figure shows how* Cad *works. Data items shown in grey are durable. In (i), the baseline is fully asynchronous; in (ii), the baseline synchronously replicates but asynchronously persists. At first, when item* a *is durable, read(*a*) passes the durability check. Items* b *and* c *are not yet durable. The check for read(*b*) fails; hence, the leader makes the state durable after which it serves* b*.*

immediately; otherwise, the leader takes additional steps (described next) to make the data durable.

The leader also periodically (via the heartbeats and replication requests) informs the followers of the durable-index. If the read arrives at a follower, it performs the same durability check (using the durable-index sent

by the leader). If the durability check passes, the follower serves the read; otherwise, it redirects the request to the leader which then makes the data durable as we describe next. Note that the durable-index might be lagging in the followers when compared to the leader. If the durable-index is stale, then the follower might consider a durable item to be non-durable and redirect that request to the leader. Therefore, this staleness does not affect the correctness of CAD.

An alternative to read-triggered durability while still ensuring only durable items are read would be to maintain two versions for each item: a durable version and a non-durable one. The system would then only serve durable versions on reads. One shortcoming of this approach is that the system needs to maintain two versions for each data item, resulting in space overheads. More importantly, the alternative approach also increases the amount of stale data that the system exposes compared to the baseline (eventually durable system). For example, if a read arrives at a node that already has the latest data, the read would see the latest data in the baseline. However, if CAD maintains two versions for each item and serves a durable version that is stale, it would increase the staleness exposed by CAD. Our goal is to improve the guarantees over the baseline; therefore, we chose to reject this design and use read-triggered durability instead which we describe next.

4.3.6 Read-triggered Durability

If the durability check fails, CAD needs to make the state (up to the latest update to the item being read) synchronously durable before serving the read. The leader treats the read for which the check fails specially. First, the leader synchronously replicates all updates up to the update-index of the item being read if these updates have not yet been replicated. The leader also informs the followers that they must flush their logs to disk before responding to this request.

When the followers receive such a request, they synchronously append the updates and flush the log to disk and respond. During such a flush, all previous writes buffered are also written to disk, ensuring that the entire state up to the latest update to the item being read is durable. Fortunately, the periodic background flushes reduce the amount of data that needs to be written during such foreground flushes. The persisted-index reported by a node as a response to this request is at least as high as the update-index of the item. When the flush finishes on a majority, the durable-index will be updated, and thus the data item can be served. The second column of Figure 4.4 shows how this procedure works. As shown, the durability check fails when item *b* is read; the nodes thus flush all updates up to index 2 and so the durability-index advances; the item is then served.

As an optimization, CAD also persists writes that are after the last update to the item being read. Consider the log [a, b, c] in Figure 4.4; when a client reads b, the durability check fails. Now, although it is enough to persist entries up to b, CAD also flushes update c, obviating future synchronous flushes when c is read as shown in the last column of the figure.

To summarize, CAD makes data durable upon reads and so guarantees that state that has been read will never be lost even if servers crash and recover.

4.3.7 Correctness

For correctness, CAD must always recover state up to the last update that was read by clients. As long as the current leader is functional, the leader ensures that the data is persisted on at least a majority before serving reads. However, CAD must be careful about how it recovers state when the current leader fails.

When the current leader fails, CAD must ensure that latest state that was read by clients survives into the new view. We argue that this is

ensured by how elections work in CAD (and in many majority-based systems). Let us suppose that the latest read has seen state up to index L. When the leader fails and subsequently a new view is formed, the system must recover all entries at least up to L for correctness; if not, an older state may be returned in the new view. The followers, on a leader failure, become candidates and compete to become the next leader. A candidate must get votes from at least a majority (may include self) to become the leader. When requesting votes, a candidate specifies the index of the last entry in its log. A responding node compares the incoming index (P) against the index of the last entry in its own log (Q). If the node has more up-to-date data in its log than the candidate (i.e., $Q > P$), then the node does *not* give its vote to the candidate. This is a property ensured by many majority-based systems [12, 19, 120] which CAD preserves.

Because CAD persists the data on at least a majority before serving reads, at least one node in any majority will contain state up to L on its disk. Thus, only a candidate that has entries at least up to L can get votes from a majority and become the leader. In the new view, the nodes follow the new leader's state. Given that the leader is guaranteed to have entries at least up to L, all data that have been served so far will survive into the new view, thus ensuring that clients will always be able to read what they read previously.

4.4 Implementation

We have built CAD by modifying ZooKeeper (v3.4.12). We have two baselines. First, ZooKeeper with synchronous replication but asynchronous persistence (i.e., ZooKeeper with *forceSync* disabled). Second, ZooKeeper with asynchronous replication; we modified ZooKeeper to obtain this baseline.

In ZooKeeper, write operations either create new key-value pairs or

update existing ones. As we discussed, CAD follows the same code path of the baseline for these operations. In addition, CAD replicates and persists updates constantly in the background. Read operations return the value for a given key. On a read, CAD performs the durability check (by comparing the key's update-index against the system's durable-index) and enforces durability if required.

CAD incurs little metadata overheads compared to unmodified ZooKeeper to perform the durability check. Specifically, ZooKeeper already maintains the last-updated index for every item (as part of the item itself [21]) which CAD reuses. Thus, CAD needs to additionally maintain only the durable-index, which is 8 bytes in size. However, some systems may not maintain the update indexes; in such cases, CAD needs eight additional bytes for every item compared to the unmodified system, a small price to pay for the performance benefits.

Performing the durability check is simple in ZooKeeper because what item a request will read is explicitly specified in the request. However, doing this check in a system that supports range queries or queries such as "get all users at a particular location" may require a small additional step. The system would need to first tentatively execute the query and determine what all items will be returned; then, it would enforce durability if one or more items are not durable yet.

We modified ZooKeeper's replication requests and responses as follows. The followers include the persisted-index in their response and the leader sends the followers the durable-index in the requests or heartbeats.

4.5 Evaluation

In our evaluation, we seek to analyze and understand the following aspects of CAD:

- CAD does not change the path of writes when compared to eventual

durability; therefore, CAD should match the performance of eventual durability on writes. Thus, we first ask: how does CAD perform during a write-only workload compared to immediate and eventual durability? (§4.5.1)

- CAD enforces durability on reads. If a read accesses a non-durable data item, CAD has to make data items durable through synchronous operations in the critical path of reads. Thus, we next ask: how much overheads does CAD incur on reads for workloads consisting of a mix of read and writes? How does this affect the overall performance of CAD? (§4.5.2)
- Finally, we ask: does CAD's implementation in ZooKeeper ensure durability of items that have been read in the presence of failures? (§4.5.3)

We conduct a set of experiments to answer these questions. We run our performance experiments with five replicas. Each replica is a 20-core Intel Xeon CPU E5-2660 machine with 256 GB memory running Linux 4.4 and uses a 480-GB SSD to store data. The replicas are connected via a 10-Gbps network. Numbers reported are the average over five runs.

We compare CAD against immediate and eventual durability. With immediate durability, the system replicates and persists writes (using *fsync*) on a majority in the critical path; it employs batching to improve performance. There are two asynchronous configurations of eventual durability and CAD: fully asynchronous and asynchronous persistence. In the fully asynchronous configuration, the system performs both replication and persistence asynchronously. With asynchronous persistence, the system replicates synchronously but persists asynchronously. Unless specified otherwise, we use the fully asynchronous configuration in our experiments.

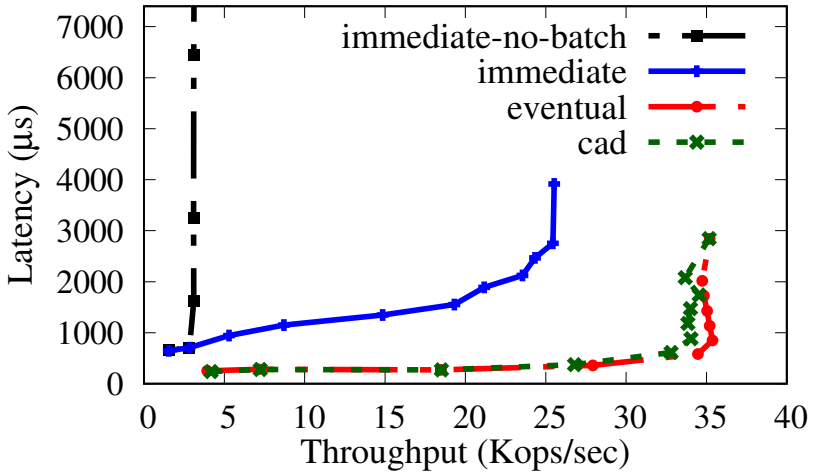
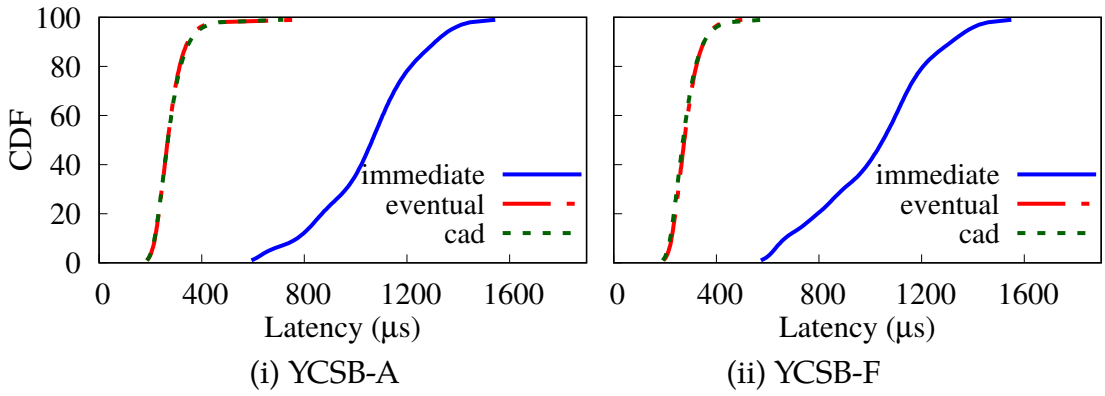


Figure 4.5: **Write-only Workload: Latency vs. Throughput.** *The figure plots the average latency against throughput by varying the number of clients for a write-only workload for different durability layers.*

4.5.1 Write-only Micro-benchmark

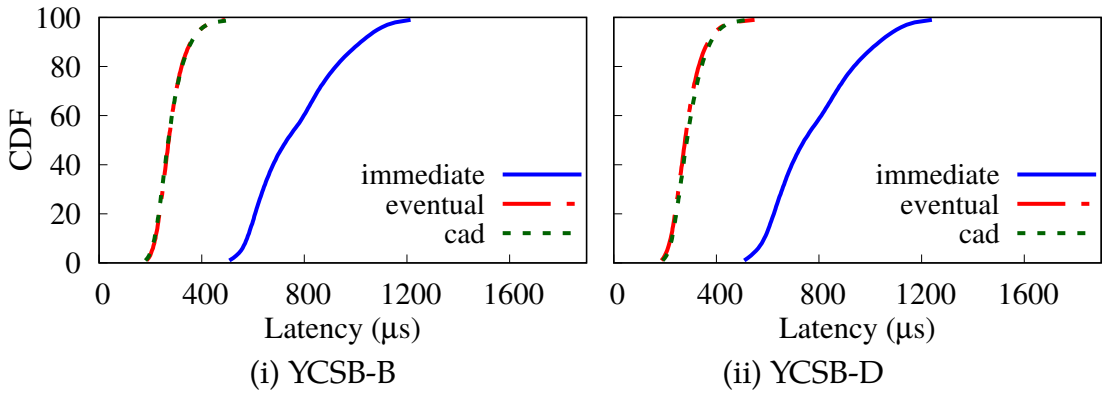
We first compare the performance for a write-only workload. Intuitively, CAD should outperform immediate durability and match the performance of eventual durability for such a workload. Figure 4.5 shows the result: we plot the average latency seen by clients against the throughput obtained when varying the number of closed-loop clients from 1 to 100. We show two variants of immediate durability: one with batching and the other without. We show the no-batch variant only to illustrate that it is too slow and we do *not* use this variant for comparison; throughout our evaluation, we compare only against the optimized immediate-durability variant that employs batching.

We make the following three observations from the figure. First, immediate durability with batching offers better throughput than the no-batch variant; however, even with aggressive batching across 100 clients, it cannot achieve the high throughput levels of CAD. Second, writes incur significantly lower latencies in CAD compared to immediate durability; for



(i) YCSB-A (ii) YCSB-F

(a) Write-heavy workloads



(i) YCSB-B (ii) YCSB-D

(b) Read-heavy workloads

Figure 4.6: **YCSB Write Latencies.** *(a)(i) and (a)(ii) show the write latency distributions for the three durability layers for write-heavy YCSB workloads. (b)(i) and (b)(ii) show the same for read-heavy YCSB workloads.*

instance, at about 25 Kops/s (the maximum throughput achieved by immediate durability), CAD's latency is 7× lower. Finally, CAD's throughput and latency characteristics are very similar to that of eventual durability.

4.5.2 YCSB Macro-benchmarks

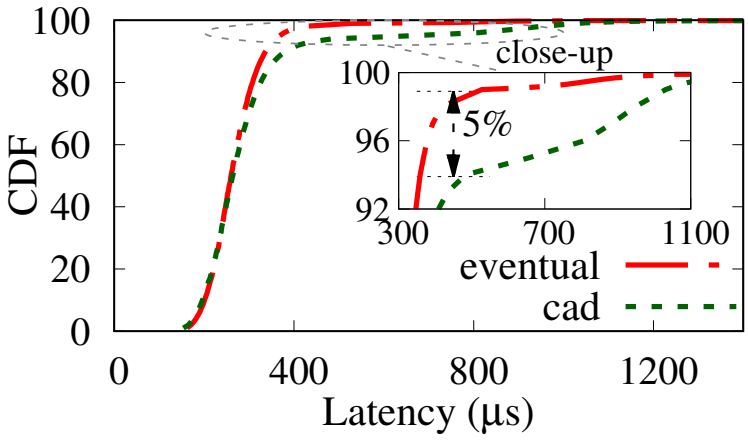
We now compare the performance across different workloads in the Yahoo! Cloud Serving Benchmark (YCSB) [44]. We use four YCSB work-

loads that have different read-write ratios and access patterns: A (w:50%, r:50%), B (w:5%, r:95%), D (read latest, w:5%, r:95%), F (read-modify-write:50%, r:50%). We do not run YCSB-E because ZooKeeper does not support range queries. The workloads in YCSB have characteristics similar to real-world applications. Workload-A reflects a session store; B resembles a photo-tagging application; D models user status updates; F reflects the read-modify-write pattern frequently used in data stores. A, B, and F have a zipfian access pattern (most operations access popular items); D has a latest access pattern (most reads are to recently modified data).

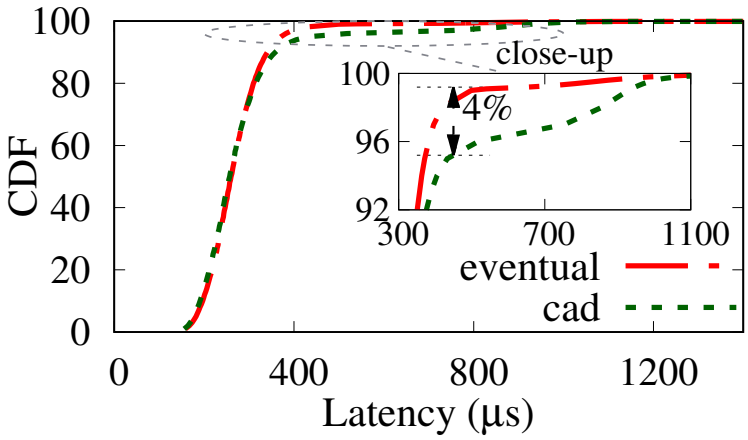
We run this experiment with 10 clients. We restrict the reads only to the leader. CAD must ideally match the performance of eventual durability and perform faster than immediate durability. We first compare the write latencies of the different durability models. We then show the overheads that CAD incurs on reads when compared to eventual durability. Finally, we compare the overall performance of the three durability models.

Write Latencies. The performance of writes in CAD should be identical to eventual durability; making data durable on reads should not affect writes. Figure 4.6 shows the latency distribution of writes for the three durability layers for different YCSB workloads that have a mix of reads and writes. As shown, writes in CAD are much faster than immediate durability. Also, writes in CAD match the performance of eventual durability for both write-heavy and read-heavy workloads.

Read Latencies. Most read operations in CAD must experience latencies similar to reads in eventual durability. However, reads that access non-durable items may trigger synchronous replication and persistence, causing a reduction in performance. This effect can be seen in the read latency distributions shown in Figures 4.7 and 4.8. As shown, a fraction of reads (depending upon the workload) trigger synchronous replication



(a) YCSB-A



(b) YCSB-F

Figure 4.7: **YCSB Read Latencies: Write-heavy workloads.** *(a) and (b) show read latencies for eventual durability and* Cad *for write-heavy YCSB workloads. The annotation within a close-up shows the percentage of reads that trigger synchronous durability in* Cad.

and persistence, and thus incur higher latencies. However, as shown in the close-ups in Figures 4.7 and 4.8, for the variety of workloads in YCSB, this fraction is small (less than 5%).

A bad workload for Cad is one that predominantly reads recently written items. Even for such a workload, the percentage of reads that actu-

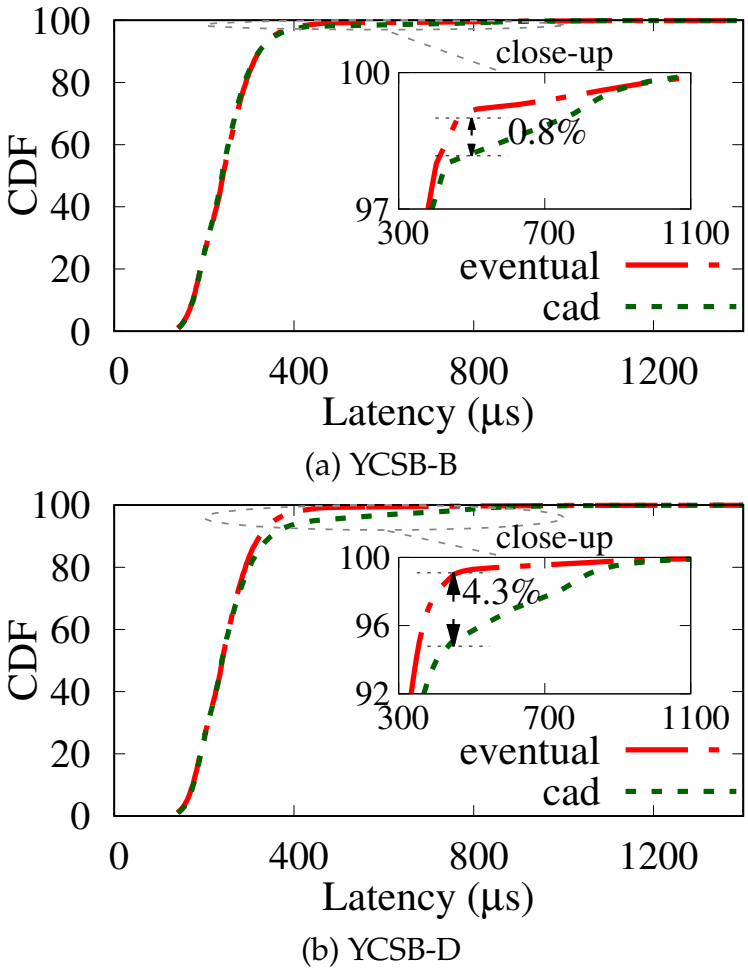
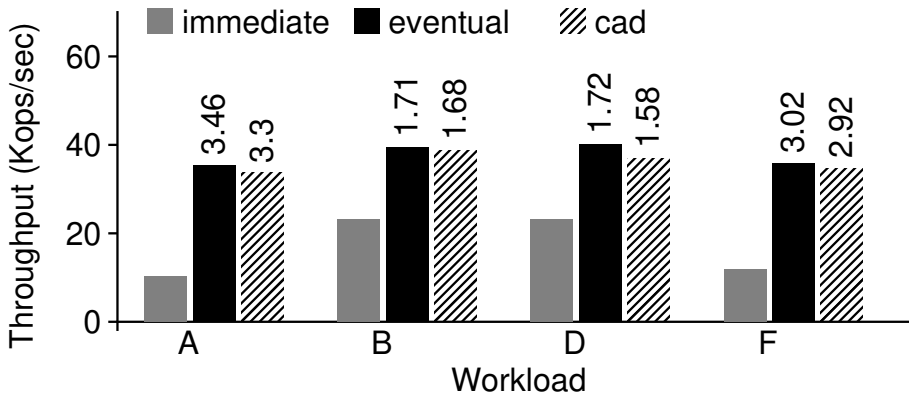


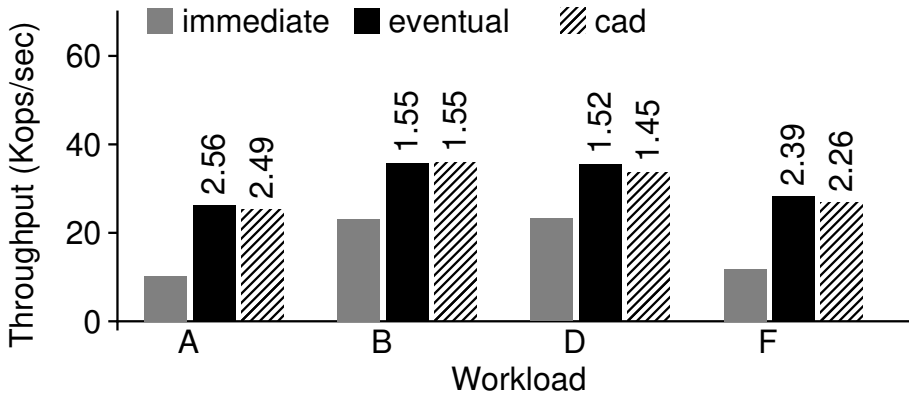
Figure 4.8: **YCSB Read Latencies: Read-heavy workloads.** *(a) and (b) show read latencies for eventual durability and* Cad *for read-heavy YCSB workloads.*

ally trigger immediate durability is small due to prior reads that make state durable and periodic background flushes in Cad. For example, with YCSB-D, although 90% of reads access recently written items, only 4.32% of these requests trigger synchronous replication and persistence (as shown in Figure 4.8 (b)).

Overall Performance. We now compare the throughput of the three durability layers for the various YCSB workloads. Figure 4.9 shows the results.



(a) Baseline: async replication & persistence



(b) Baseline: async persistence

Figure 4.9: **Overall Performance.** *The figure compares the throughput of the three durability layers. In (a), eventual and* CAD *are fully asynchronous; in (b), they replicate synchronously but persist lazily. The number on top of each bar shows the factor of improvement over immediate durability.*

In Figure 4.9(a), eventual and CAD carry out both replication and persistence asynchronously; whereas, in 4.9(b), they replicate synchronously but persist to storage lazily.

As shown in Figure 4.9(a), compared to immediate durability with batching, CAD's performance is significantly better. CAD is about 1.6× and 3× faster than immediate durability for read-heavy workloads (B and D)

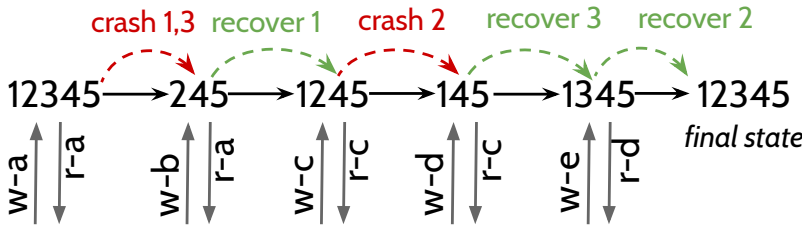


Figure 4.10: **An Example Failure Sequence.** *The figure shows an example sequence generated by our test framework.*

and write-heavy workloads (A and F), respectively. CAD closely matches the performance of eventual durability even in the presence of some reads that trigger synchronous durability in CAD. For the different YCSB workloads, the drop in performance for CAD compared to eventual durability is little (2% – 8%). Even, for the worst-case read-latest workload (YCSB-D), CAD's overhead compared to eventual durability is only 8%. Similar results and trends can be seen for the asynchronous-persistence baseline in Figure 4.9(b).

4.5.3 Durability Guarantees

We now check if CAD's implementation correctly ensures durability of read data items in the presence of failures. To do so, we developed a framework that can drive the cluster to different states by injecting crash and recovery events. Figure 4.10 shows an example sequence. At first, all nodes are alive; then nodes 1, 3 crash; 1 recovers; 2 crashes; 3 recovers; finally, 2 recovers. In addition to crashing, we also randomly choose a node and introduce delays to it; such a lagging node may not have seen a few updates. For example, $\boxed{1}2345 \rightarrow 245 \rightarrow 1\boxed{2}45 \rightarrow 145 \rightarrow 134\boxed{5} \rightarrow 12345$ shows how nodes 1, 2, and 5 experience delays in a few states.

At each intermediate state, the framework inserts new items and reads a few items. For instance, in Figure 4.10, item a is written and read back

Read type	Durability	Outcomes (%)		
		Correct	Data loss	Read-data loss
random	immediate	100	0	0
	eventual	50	50	39
	CAD	87	13	0
latest	immediate	100	0	0
	eventual	50	50	50
	CAD	100	0	0

Table 4.2: **Durability.** *The table shows the durability-experiment results for the three durability models.*

in the first state; b is written and a is read again in the second state and so on. At the end, the framework recovers all nodes and issues reads to all inserted items. If any item is lost, we flag the sequence as *data loss*; if any previously *read* item is lost, we additionally flag the sequence as *read-data loss* (a subset of the general data-loss case). Using the framework, for a five-node cluster, we generated 500 random sequences similar to the one in Figure 4.10. We first subject the unmodified versions of ZooKeeper to the sequences; then, we do the same for CAD.

Table 4.2 shows the results. In this experiment, eventually durable ZooKeeper and CAD persist asynchronously but replicate synchronously. Our first workload issues reads to a randomly chosen set of acknowledged items; we call these reads "random".

ZooKeeper with immediate durability, as expected, does not lose any data in any of the 500 sequences. With asynchronous persistence, ZooKeeper loses data in 50% of cases and more importantly, loses data that has been read by the clients in 39% of cases. CAD, in contrast, never loses data that

has been read: state that has been made visible on reads is always recovered (0% read-data loss). CAD loses some (unread) items in some cases; this is the basic tradeoff that CAD makes, delaying the durability of data that has not been read for high performance. However, compared to eventually durable ZooKeeper, CAD significantly reduces the data loss cases (from 50% to 13%) due to its periodic flushes and read-triggered durability. We note that the numbers do not denote the likelihood of data loss; rather, they reflect what happens when the system is stressed.

CAD can recover *all* written data, irrespective of failures, if a request successfully reads the last-inserted item. To show this aspect, we conduct a slightly different experiment in which we use the same crash sequences but perform a read of the last-inserted item at each intermediate state ("latest"). In such cases, as shown in the second part of the table, eventually durable ZooKeeper loses data in about 50% of cases; in contrast, CAD recovers all data, avoiding any data-loss instances, behaving similar to immediately durable ZooKeeper.

4.5.4 Summary

CAD is significantly faster than immediate durability (that is optimized with batching) while matching the performance of eventual durability for many workloads. Even for workloads that mostly read recently modified items, CAD's overheads are small. Our implementation of CAD ensures that data that has been read by clients is always durable.

4.6 Implementing CAD in Redis

So far, we discussed how we implemented and evaluated CAD in ZooKeeper. We now demonstrate that CAD applies to other systems as well. We also show that implementing CAD requires only moderate developer effort, requiring minimal code changes. To this end, we implement CAD in Redis,

another widely used leader-based system. In this section, we first provide an overview of Redis and then describe our implementation. We then evaluate the performance of our implementation and compare it against baseline Redis.

4.6.1 Redis Overview

Redis is a popular leader-based data structure store. Clients submit write requests to the leader which appends the update to an on-disk append-only file and then replicates the update to the followers. Similar to ZooKeeper, Redis also has two baselines. In the fully asynchronous baseline, Redis performs both replication and persistence asynchronously, i.e., updates are acknowledged immediately after they have been buffered in memory on the leader; this is the default configuration in Redis. We also configure Redis to replicate synchronously (using the `WAIT` option [139]) but persist asynchronously to obtain the second baseline. In both the baselines, Redis issues *fsync* in the background periodically (every second).

CAD-Redis follows the same code path of the baseline for updates. However, upon reads, it performs the durability check and enforces durability if necessary before serving reads.

We compare the baselines and CAD-Redis against an immediately durable version of Redis which performs both replication and persistence synchronously. We obtain this immediately durable configuration by setting appropriate values for the `WAIT` and the `appendfsync` options.

4.6.2 Redis Implementation

We implement CAD in Redis v4.0.11. Native Redis does not perform automatic failover (i.e., if the current leader fails, it does not elect a new leader automatically). Thus, we use Redis with sentinel [138] which enables automatic failover. Our implementation took only a moderate developer

effort: we added or changed less than 1K LOC in Redis. Further, implementing CAD required little changes to the rest of the system.

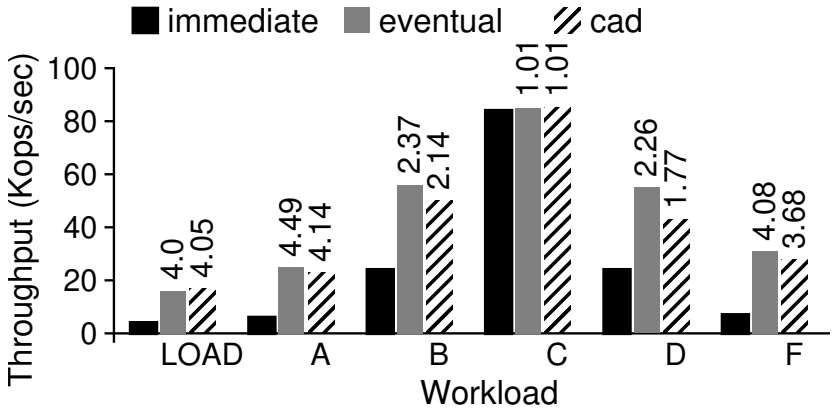
Compared to the ZooKeeper implementation, two additional changes were required. First, we added new structures to quickly lookup the *update-index* of an item because Redis does not have such a structure unlike ZooKeeper. Upon reads, CAD-Redis looks up this structure to perform the durability check. Second, Redis does not ensure that the leader always has all the committed data unlike ZooKeeper, i.e., a node that has not seen some updates may be elected the leader. This is because although the sentinel requires a majority vote to choose the new leader, it does not take into account the node's last log index. We thus modified the leader election to take a node's last log index into consideration during election. This modification ensures that the chosen node has all the data that has been read so far, thus ensuring the correctness of CAD.

4.6.3 Performance

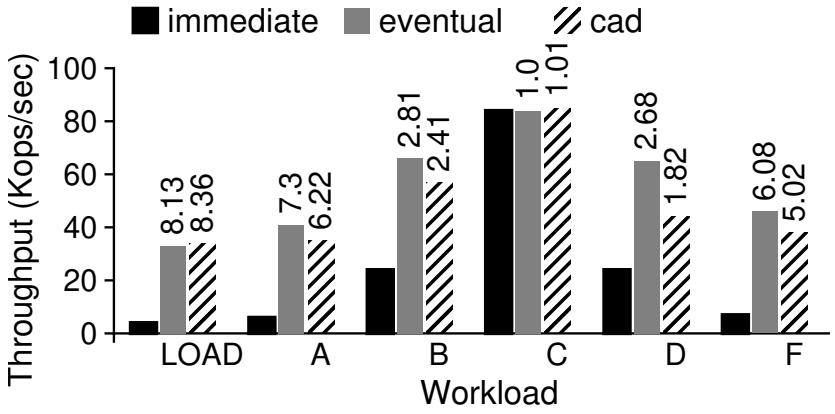
We now evaluate the performance of CAD in Redis. In addition to the workloads described in §4.5, we also run workloads LOAD and C; LOAD is a write-only workload and C is a read-only workload. Similar to ZooKeeper, we compare the CAD version of Redis (CAD-Redis) against immediate-Redis (that uses synchronous replication and synchronous persistence) and eventual-Redis. Figure 4.11 shows the result. In (a), eventual-Redis and CAD-Redis perform both replication and persistence lazily; in (b), eventual-Redis and CAD-Redis perform replication synchronously but persist to storage lazily.

Similar to ZooKeeper, CAD offers performance benefits when compared to immediate durability in Redis as well. Also, CAD closely matches the performance of eventual durability in Redis for most workloads.

As shown in Figure 4.11(a), for write-heavy (load, A, and F) and read-heavy (B) workloads, CAD-Redis is notably faster (2.14×-4.14×) than immediate-Redis and adds only little overhead when compared to eventual-Redis



(a) Baseline: async persistence



(b) Baseline: async replication & persistence

Figure 4.11: **Redis Performance.** *The figure compares the throughput of immediate, eventual, and* CAD *durability layers in Redis. In (a), eventual and* CAD *synchronously replicate but asynchronously persist; in (b), they replicate and persist lazily. The number on top of each bar shows the performance normalized to that of immediate durability.*

(about 10% lower throughput). In the worst-case read-latest workload (D), CAD-Redis offers 22% lower throughput than eventual-Redis but is still 77% faster than immediate-Redis.

Figure 4.11(b) shows the performance results when the baseline employs asynchronous replication and persistence. Compared to the case

when the baseline replicates synchronously (i.e., Figure 4.11(a)), the difference in performance between CAD-Redis and eventual-Redis is slightly higher (e.g., 14% lower throughput instead of 10% for workload-B). At the same time, CAD-Redis is significantly faster than immediate-Redis for many workloads (for example, 6.22× higher throughput instead of 4.14× for workload-A).

Overall, implementing CAD in another system was fairly straightforward, requiring only minimal code changes. Furthermore, similar to the ZooKeeper case, CAD-Redis offers significant performance benefits over immediately durable Redis and closely approximates the performance of eventually durable Redis.

4.7 Discussion

In this section, we discuss how CAD can be beneficial for current systems and deployments, and how it can be implemented in other classes of systems (e.g., leaderless ones). We then discuss how CAD offers benefits even with the advent of fast storage devices.

Application usage. As we discussed, most widely used systems lean towards performance and thus adopt eventual durability. CAD's primary goal is to improve the guarantees of such systems. By using CAD, these systems and applications atop them can realize stronger semantics without forgoing the performance benefits of asynchrony. Further, little or no modifications in application code are needed to reap the benefits that CAD offers.

A few applications such as configuration stores [65] cannot tolerate any data loss and so require immediate durability upon every write. While CAD may not be suitable for this use case, a storage system that implements CAD can support such applications. For example, in CAD, applications can optionally request immediate durability by specifying a flag in

the write request (of course, at the cost of performance). Alternatively, an application can ensure the durability of all the data it wrote so far by issuing a read to the latest written item; read-triggered durability ensures that the entire state up to the last update that modifies the item are durable.

A few applications may be write-heavy and read the latest written data items. In such cases, CAD might simply shift the cost of synchronous operations from writes to reads and might offer little or no benefits over immediate durability. Since immediate durability offer better guarantees, in such worst-case scenarios, the system could detect this workload behavior and shift to using immediate durability. We leave this dynamic switching as an avenue for future work. However, we believe that most real-world workloads do not immediately read what they wrote (similar to the YCSB workloads we evaluated in §4.5.2).

CAD for other classes of systems. While we apply CAD to leader-based systems in this paper, the idea also applies to other systems that establish no or only a causal order of updates. However, a few changes compared to our implementation for leader-based systems may be required. First, given that there is no single update order, the system may need to maintain metadata for each item denoting whether it is durable or not (instead of a single durable-index). Further, when a non-durable item x is read, instead of making the entire state durable, the system may make only updates to x or ones causally related to x durable. We leave such extension as an avenue for future work.

CAD for sharded systems. In this dissertation, we implement CAD in non-sharded storage systems. Several storage systems such as ZooKeeper, etcd, and LogCabin are used in non-sharded deployments and thus our ideas and implementation can be readily applied. However, many deployments need to use sharding to scale horizontally. We believe CAD can be used as-is or extended to suit such needs. First, if the shards are independent and do not have write dependencies across them, then CAD

can be applied as described in this chapter. When a request reads a non-durable item on a shard, only items within that shard (that the item being read depends upon) need to be made durable. However, if there are write dependencies across shards, then a read of an item on one shard may require communication with another shard to make the dependent updates durable. In such cases, we believe the leaders of the different shards can exchange information (specifically, their local durable index) in the background; thus, in many cases, one can expect the dependencies to be already durable and hence may not require communication in the synchronous path of reads.

Advent of faster storage. Our evaluation shows that CAD can offer significantly higher performance than immediate durability when using slow storage (e.g., SSDs) within the data centers. We now discuss if these benefits will still hold with the advent of faster storage (e.g., NVM).

If a system must use synchronous replication and if it uses asynchronous persistence for better performance, then using NVMs can offer better guarantees at nearly the same performance. However, CAD still offers significant benefits in the following two scenarios. First, while faster devices alleviate the costs of persistence, in the wide-area, synchronous replication will still incur large network latencies. Thus, a system that asynchronously replicates with CAD can obtain higher performance than an immediately durable system. Second, even within the data center, CAD would deliver ~2× lower latencies compared to immediate durability. Assuming NVM to be as fast as DRAM, the second row of Table 4.1 approximately represents the performance of a system that synchronously replicates and synchronously persists to NVMs; in such cases, asynchronous replication with CAD would offer 2.4× higher performance as shown in Table 4.1.

4.8 Summary and Conclusions

In this chapter, we show how the underlying durability model of a distributed system has strong implications for its consistency and performance. We present consistency-aware durability (CAD), a new approach to durability in distributed storage systems. CAD shifts the point the system makes data durable from writes to reads: data is made durable before it is made visible to clients upon reads. By delaying durability of writes, CAD achieves high performance. However, by guaranteeing the durability of data that has been read, CAD enables stronger consistency. In the next chapter, we discuss how to realize stronger consistency atop CAD with high performance.

While enabling stronger consistency, CAD may not be suitable for a few applications that cannot tolerate any data loss. However, it offers a new, useful middle ground for many systems that currently use eventual durability to realize stronger semantics without compromising on performance.

5

Building Strong Consistency upon Consistency-aware Durability

In the previous chapter, we introduced *consistency-aware durability* (CAD), a new durability model for distributed systems that can enable stronger consistency with high performance. In this chapter, we discuss how we realize one such stronger consistency property that we refer to as *cross-client monotonic reads* atop CAD. Cross-client monotonicity cannot be realized efficiently without a consistency-aware layer: immediate durability can enable it but is slow; on the other hand, it simply cannot be realized upon eventual durability.

Cross-client monotonic reads guarantees that a read from a client will return a state that is at least as up-to-date as the state returned to a previous read from any client. Cross-client monotonic reads provide these guarantees even in the presence of failures and across client sessions. Such guarantees can be useful for applications such as location sharing, social media timelines, and shopping carts. This property can be also beneficial in edge-computing scenarios, where clients may reconnect to the storage system often because of frequent disconnections or mobility; in such scenarios, cross-client monotonicity can provide strong guarantees across client sessions.

CAD, while necessary, is not sufficient if the system needs to ensure cross-client monotonicity while allowing reads at many replicas. To this

end, we introduce *active sets*, a lease-based technique that ensures monotonic reads while allowing reads at many nodes. With active sets, the system can permit clients to read at nearby replicas, making it well suited for geo-replicated settings.

We design and implement cross-client monotonic reads upon the CAD version of ZooKeeper to build ORCA. We experimentally show that ORCA provides strong guarantees while closely matching the performance of weakly consistent ZooKeeper. Compared to strongly consistent ZooKeeper, ORCA provides significantly higher throughput (1.8 – 3.3×), and notably reduces latency, sometimes by an order of magnitude in geo-distributed settings. This chapter is based on the later parts of the paper, *Strong and Efficient Consistency with Consistency-Aware Durability*, published in FAST 20 [60].

The chapter is organized as follows. We first discuss the tradeoffs between performance and consistency in distributed systems (§5.1). We then discuss how CAD helps overcome this tradeoff and introduce our new consistency model, *cross-client monotonic reads* (§5.2). Next, we describe the design and implementation of ORCA (§5.3). Next, we present our evaluation (§5.4). We also demonstrate how the guarantees provided by ORCA can be useful in two application scenarios (§5.5). Finally, we summarize and conclude (§5.6).

5.1 Consistency vs. Performance

As we discussed in the previous chapter (§4.1), linearizability or strong consistency is expensive. A key reason linearizability or strong consistency incurs high overheads is that it often requires immediate durability; preventing stale and out-of-order reads require data to be durable in the critical path of writes. These synchronous writes make a linearizable system too slow: within a data center, immediate durability is about 10×

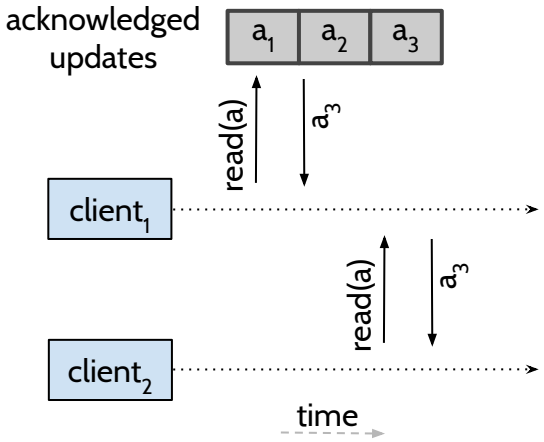


Figure 5.1: **Linearizability.** *The figure shows possible values clients can observe upon reads in a linearizable system. The system has acknowledged updates* a_1, a_2, *and* a_3. *Time flows from left to right.*

slower than asynchronous durability.

Immediate durability, while necessary, is not sufficient to prevent out-of-order and stale reads; additional mechanisms are required. For example, consider a linearizable system [92] that synchronously persists an update on a majority. In such a system, it is possible for a minority of followers to have not seen this update and so be lagging. Without additional mechanisms, a client might read the updated state that contains the latest state from the leader and a later request from the same or a different client might notice an older state if it reads from the lagging followers, violating linearizability. A few linearizable systems, to prevent such situations, allow reads only at the leader [80, 92, 108, 119]. Also, the leader before serving a read contacts a majority to check if it is deposed by a new leader [†]. These mechanisms enable a linearizable system to prevent out-of-order and stale reads.

Linearizability not only offer strong guarantees within a single client

[†]Additional mechanisms (e.g., duplicate request filtering [87]) are required for linearizability but immediate durability and leader restriction are the ones that affect the performance the most.

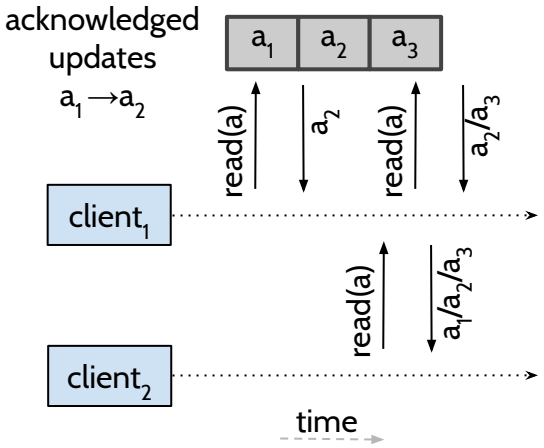


Figure 5.2: **Causal Consistency.** *The figure shows possible values clients can observe upon reads in a causally consistent system. The system has acknowledged updates* a_1, a_2, *and* a_3*; updates* a_1 *and* a_2 *are causally related.*

session but also across clients. For instance, consider the example shown in Figure 5.1, where the storage system has acknowledged updates a_1, a_2, and a_3. A linearizable system will not serve a client an updated state (a_3) at one point and subsequently serve an older state (a_1 or a_2) to *any* client. Further, when a client crashes and recovers, it is guaranteed to read what it read in its previous session or a later state (i.e., it won't go back in time).

However, restricting reads to one node severely limits read throughput; further, it prevents clients from reading from their nearest replica, increasing read latencies (especially in geo-distributed settings where clients have to incur wide-area latencies to reach the leader). Consequently, practical systems allow reads at many nodes [103, 107, 120, 140] (at the cost of strong consistency).

While linearizability is expensive, weaker models such as causal consistency, monotonic reads, and eventual consistency are performant. This is because weakly consistent systems are often built atop performant eventual durability (as discussed earlier). However, a weakly durable system can expose stale and out-of-order reads. For example, an application may

read a data item that is buffered on the memory of a server; upon a failure of the server, in a *different* client session, the application can read from the same server and notice that the item is lost (the server lost data buffered in memory during the failure), exposing out-of-order states. However, this scenario does not violate guarantees such as causal consistency and monotonic reads; while these consistency models provide in-order reads, they do so only within a *single* client and not across clients. Consider the example shown in Figure 5.2, where the storage system has acknowledged updates a_1, a_2, and a_3 and updates a_1 and a_2 are causally related. While a causally consistent system will not serve a_2 to $\texttt{client}_1$ at one point and subsequently serve a_1 to the same client ($\texttt{client}_1$), it can serve a_1 to another client ($\texttt{client}_2$).

Weakly consistent systems can expose non-monotonic states also because they usually allow reads at many nodes [40]. For example, a client can reconnect to a different server after a disconnection, and may read an older state in the new session if a few updates have not been replicated to this server yet. For the same reason, two client sessions to two different servers from a single application may receive non-monotonic states. While the above cases do not violate causal consistency by definition (because it is a different client session), they lead to poor semantics for applications.

5.2 Stronger and Efficient Consistency with CAD

We now discuss how the seemingly conflicting goals of strong consistency and high performance can be realized together in a storage system upon a durability layer that is consistency-aware. We first describe the new consistency property. We then discuss its utility and finally discuss the importance of allowing reads at many nodes.

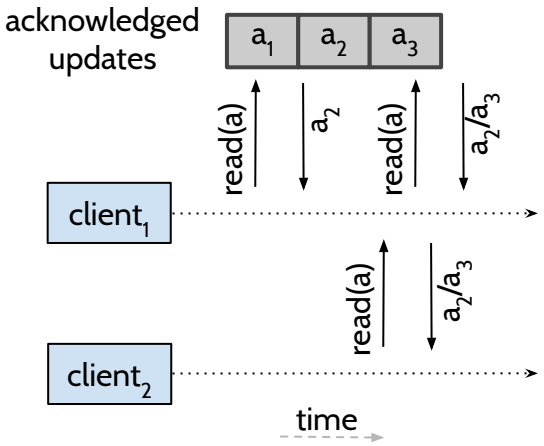


Figure 5.3: **Cross-client Monotonic Reads.** *The figure shows possible values clients can observe upon reads under our new consistency model. The system has acknowledged updates* a_1, a_2, *and* a_3.

5.2.1 Cross-client Monotonic Reads and CAD

We first observe that eventual durability can lose data arbitrarily upon failures, and so prevents the realization of both non-stale and monotonic reads together. While preventing staleness requires expensive immediate durability upon every write, we note that monotonic reads across failures can be useful in many scenarios and can be realized efficiently upon a consistency-aware durability (CAD) layer we introduced in the previous chapter.

To this end, we introduce *cross-client monotonic reads*, a new consistency property that offers strong guarantees. This property guarantees that a read from a client will always return a value that is at least as up-to-date as the value returned to a previous read from *any* client, irrespective of server and client failures, and across sessions. Figure 5.3 shows possible values clients can observe under this property. As shown, once $\texttt{client}_1$ observes a_2, $\texttt{client}_1$ and $\texttt{client}_2$ should observe at least a_2.

While linearizability provides this property, it does so at the cost of performance; immediate durability and leader-restricted reads make a

linearizable system too expensive. Weaker consistency models built atop eventual durability cannot provide this property. Even sequential consistency which is weaker than linearizability but stronger than models such as causal consistency does not provide this property. Note that cross-client monotonicity is a stronger guarantee than the traditional monotonic reads that ensures monotonicity only within a session and in the absence of failures [23, 96, 170].

We realize cross-client monotonic reads efficiently upon CAD. Recall that the key idea behind CAD is to shift the point of durability to reads from writes. By allowing writes to be completed asynchronously, CAD achieves high performance. However, by enforcing durability before reads i.e., by ensuring that data is replicated and persisted before it is read by clients, CAD enables monotonic reads even across failures. Cross-client monotonicity cannot be realized efficiently without a consistency-aware layer: immediate durability can enable it but is slow; it simply cannot be realized upon eventual durability as it can lose data items that have been read upon failures.

We note that cross-client monotonic reads does not prevent staleness. However, it avoids exposing out-of-order states to applications and can be useful in many scenarios, as we discuss next.

5.2.2 Utility of Cross-client Monotonic Reads

We observe that out-of-order states provide confusing semantics for many real-world applications. As a simple example, consider the view count of a video hosted by a service; such a counter should only increase monotonically. However, in a system that can lose data that has been read, clients can notice counter values that may seem to go backward. As another example, in a location-sharing service, it might be possible for a user to incorrectly notice that another user went backwards on the route, while in reality, the discrepancy is caused by the underlying storage sys-

tem that served the updated location, lost it, and thus later reverted to an older one. Or, consider a social-media application, where users see some posts in their timelines only to go back and not find those posts again. A system that offers cross-client monotonic reads avoids such cases under all scenarios, providing better semantics.

On the other hand, consider a system that provides causal consistency. Such a system would ensure that causally related operations are seen in order across client sessions: a client observes an effect only after observing the cause of the effect. For example, consider the social-media application we discussed above. If the application is built upon a causally consistent system, a user will never see a response to a post before seeing the post itself. Or, if a user X posts a status after they change their privacy settings to exclude another user Y from viewing their posts, Y will never see X's later posts. Causal consistency would also ensure that users see posts in order within a single client session; for example, if a user sees a comment to a post in a session and views the post again in the same session, the user will at least see the comments they saw earlier. However, causal consistency does not ensure monotonic reads across client sessions. For example, a user could see the comment (effect) and its post (cause) in one session and in a later session see neither or the post alone (although, causal consistency would prevent the case where the comment alone is seen). Similarly, in the privacy-setting example, with causal consistency, user Y can notice that they cannot view X's posts in a session; then, in a later session, Y can notice that they can view X's older posts (causal consistency would prevent Y from seeing X's later posts). A system that offers cross-client monotonic reads would provide better semantics by avoiding these out-of-order states across the client sessions.

5.2.3 Need for Scalable Cross-client Monotonic Reads

To ensure cross-client monotonic reads, most existing linearizable systems restrict reads to the leader, affecting scalability and increasing latency. In contrast, a system that provides cross-client monotonic reads while allowing reads at multiple replicas offers attractive performance and consistency characteristics in many use cases. First, it distributes the load across replicas and enables clients to read from nearby replicas, offering low-latency reads in geo-distributed settings. Second, similar to linearizable systems, it provides monotonic reads, irrespective of failures, and across clients and sessions which can be useful for applications at the edge [110]. Clients at the edge may often get disconnected and connect to different servers (for example, due to user mobility), but still can get monotonic reads across these sessions.

5.3 ORCA Design

We now describe Orca, a leader-based majority system that implements cross-client monotonic reads upon Cad. We use a weakly consistent system built atop eventual durability as the baseline to highlight how Orca is different from it. Orca aims to perform similarly to this baseline but enable stronger consistency. We first outline Orca's guarantees (§5.3.1). Next, we explain how we realize cross-client monotonic reads while allowing reads at many nodes (§5.3.2). Finally, we explain how Orca correctly ensures cross-client monotonic reads (§5.3.5) and describe our implementation (§5.3.6).

5.3.1 Guarantees

As discussed earlier (in §2.3), in leader-based systems, a leader establishes a single order of updates. Orca preserves the properties of a leader-based

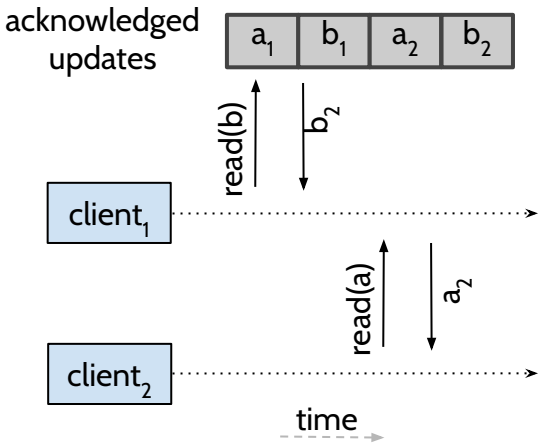


Figure 5.4: **Orca Guarantees.** *The figure shows possible values clients can observe upon reads with* Orca. *The system has acknowledged updates* a_1, b_1, a_2, *and* b_2. *Once* $client_1$ *notices* b_2, *further reads must notice all updates upto* b_2; *thus, a later read from* $client_2$ *to item* a *notices* a_2.

system that uses eventual durability, i.e., it provides sequential consistency. However, in addition, it also provides cross-client monotonic reads under all failure scenarios (e.g., even if all replicas crash and recover), and across sessions. Recall that in Cad, when a read for an item i is served, Cad guarantees that the *entire state* up to the last update that modifies i (say L) are durable. Similarly, Orca built atop Cad ensures that if reads expose an updated state (up to the last update L) at one point to a client, Orca will not later expose an older state (that does not include some updates before L) to any client, as shown in Figure 5.4. Orca is different from linearizable systems in that it does not guarantee that reads will never see stale data.

Majority-based systems remain available as long as a majority of nodes are functional [20, 120]; Orca ensures the same level of availability. Orca ensures the same level of availability as the eventually-durable baseline; for example, ZooKeeper remains available when a majority of nodes are up and connected to each other; Orca maintains the same level of avail-

ability.

5.3.2 Cross-Client Monotonic Reads with Leader Restriction

In the previous chapter, we have discussed how CAD ensures that state that has been read by clients remains durable. We now describe how upon such a durability primitive, we build cross-client monotonic reads efficiently.

If reads are restricted only to the leader, a design that many linearizable systems adopt, then cross-client monotonic reads is readily provided by CAD; no additional mechanisms are needed. Given that updates go only through the leader, the leader will have the latest data, which it will serve on reads (if necessary, making it durable before serving). Further, if the current leader fails, the new view will contain the state that was read. Thus, monotonic reads are ensured across failures.

However, restricting reads only to the leader limits read scalability and prevents clients from reading at nearby replicas. Most practical systems (e.g., MongoDB, Redis), for this reason, allow reads at many nodes [103, 107, 140]. However, when allowing reads at the followers, CAD alone cannot ensure cross-client monotonic reads. Consider the scenario in Figure 5.5. The leader S_1 has served versions a_1 and a_2 after making them durable on a majority. However, follower S_5 is partitioned and so has not seen a_2. When a read later arrives at S_5, it is possible for S_5 to serve a_1; although S_5 checks that a_1 is durable, it does not know that a has been updated and served by others, exposing non-monotonic states. Thus, additional mechanisms are needed which we describe next.

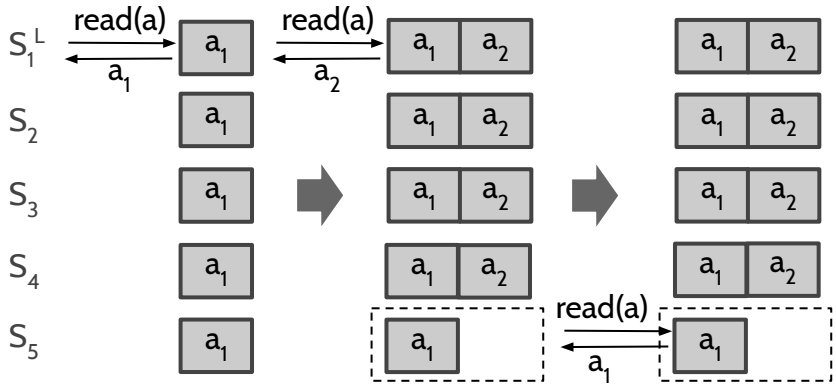


Figure 5.5: **Non-monotonic Reads.** *The figure shows how non-monotonic states can be exposed atop* CAD *when reading at the followers.*

5.3.3 Scalable Reads with Active Set

A naive way to solve the problem shown in Figure 5.5 is to make the data durable on all the followers before serving reads from the leader. However, such an approach would lead to poor performance and, more importantly, decreased availability: reads cannot be served unless all nodes are available. Instead, ORCA solves this problem using an *active set*. The active set contains *at least* a majority of nodes. ORCA enforces the following rules with respect to the active set.

R1: When the leader intends to make a data item durable (before serving a read), it ensures that the data is persisted and applied by *all* the members in the active set.

R2: Only nodes in the active set are allowed to serve reads.

The above two rules together ensure that clients never see non-monotonic states. **R1** ensures that all nodes in the active set contain all data that has been read by clients. **R2** ensures that only such nodes that contain data that has been previously read can serve reads; other nodes that do not contain the data that has been served (e.g., S_5 in Figure 5.5) are precluded from serving reads, preventing non-monotonic reads. The key challenge now is to maintain the active set correctly.

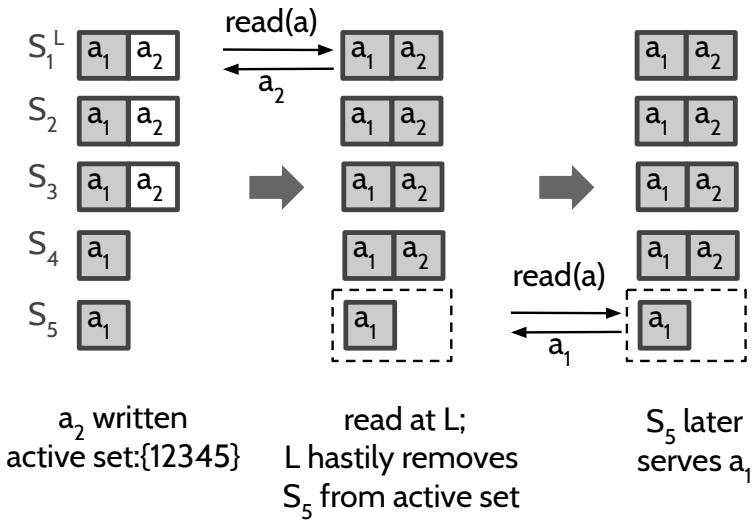


Figure 5.6: **Active Set and Leases: Unsafe Removal of Follower.** *The figure shows how if the leader removes a follower hastily then the system can expose non-monotonic states.*

5.3.4 Active Set Membership using Leases

The leader constantly (via heartbeats and requests) informs the followers whether they are part of the active set or not. The active-set membership message is a lease [31, 63] provided by the leader to the followers: if a follower F believes that it is part of the active set, it is guaranteed that no data will be served to clients without F persisting and applying the data. The lease breaks when a follower does not hear from the leader for a while. Once the lease breaks, the follower cannot serve reads anymore. The leader also removes the follower from the active set, allowing the leader to serve reads by making data durable on the updated (reduced) active set.

To ensure correctness, a follower must mark itself out *before* the leader removes it from the active set. Consider the scenario in Figure 5.6, which shows how non-monotonic states can be exposed if the leader removes a disconnected follower from the active set hastily. Initially, the active

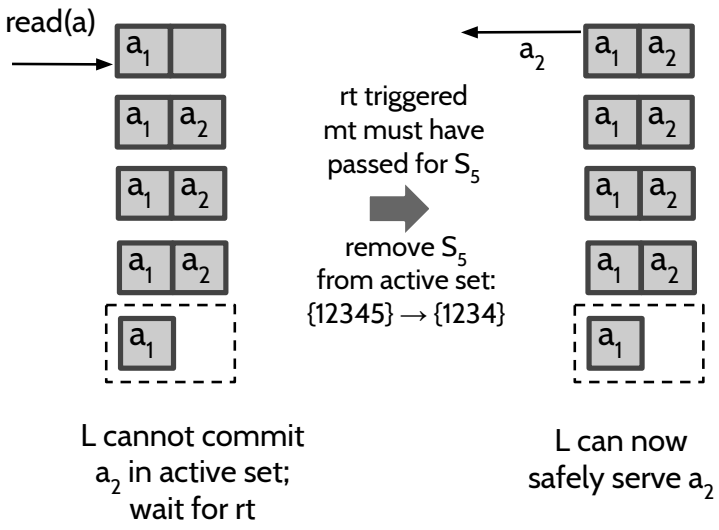


Figure 5.7: **Active Set: Two-step Breaking of Lease.** *The figure shows how* Orca *breaks leases in two steps.*

set contains all the nodes, and so upon a read, the leader tries to make a_2 durable on all nodes; however, follower S_5 is partitioned. Now, if the leader removes S_5 (before S_5 marks itself out) and serves a_2, it is possible for S_5 to serve a_1 later, exposing out-of-order states. Thus, for safety, the leader must wait for S_5 to mark itself out and then only remove S_5 from the active set, allowing the read to succeed.

Orca breaks leases using a two-step mechanism: first, a disconnected follower marks itself out of the active set; the leader then removes the follower from the active-set. Orca realizes the two-step mechanism using two timeouts: a mark-out timeout (*mt*) and a removal timeout (*rt*); once *mt* passes, the follower marks itself out; once *rt* passes, the leader removes the follower from the active set. Orca sets *rt* significantly greater than *mt* (e.g., $\mathtt{rt} >= 5 * \mathtt{mt}$) and *mt* is set to the same value as the heartbeat interval. Figure 5.7 illustrates how the two-step mechanism works in Orca. The performance impact is minimal when the leader waits to remove a failed follower from the active set. Specifically, only reads that access (re-

cently written) items that are not durable yet must wait for the active set to be updated; the other vast majority of reads can be completed without any delays.

Like any lease-based system, ORCA requires non-faulty clocks with a bounded drift [63]. By the time *rt* passes for the leader, *mt* must have passed for the follower; otherwise, non-monotonic states may be returned. However, this is highly unlikely because we set *rt* to a multiple of *mt*; it is unlikely for the follower's clock to run too slowly or the leader's clock to run too quickly that *rt* has passed for the leader but *mt* has not for the follower. In many deployments, the worst-case clock drift between two servers is as low as 30 μs/sec [61] which is far less than what ORCA expects. Note that ORCA requires only a bounded drift, not synchronized clocks.

On a read, a follower checks if it is a part of the active set. The follower then checks if the item being read is durable by comparing the update-index of the item with the durable-index (sent by the leader during heartbeats). If the durability check passes, the follower serves the read; else, it redirects the request to the leader which then makes the read durable on the active set. When a failed follower recovers (from a crash or a partition), the leader adds the follower to the active set. However, the leader ensures that the recovered node has persisted and applied all entries up to the durable-index before adding the node to the active set. Sometimes, a leader may break the lease for a follower G even when it is constantly hearing from G, but G is operating slowly (perhaps due to a slow link or disk), increasing the latency to flush when a durability check fails. In such cases, the leader may inform the follower that it needs to mark itself out and then the leader also removes the follower from the active set.

The size of the active set presents a tradeoff between scalability and latency. If many nodes are in the active set, reads can be served from them all, improving scalability; however, reads that access recently written non-

durable data can incur more latency because data has to be replicated and persisted on many nodes. In contrast, if the active set contains a bare majority, then data can be made durable quickly, but reads can be served only by a majority.

Deposed leaders. A subtle case that needs to be handled is when a leader is deposed by a new one, but the old leader does not know about it yet. The old leader may serve some old data that was updated and served by the other partition, causing clients to see non-monotonic states. Orca solves this problem with the same lease-based mechanism described above. When followers do not hear from the current leader, they elect a new leader but do so after waiting for a certain timeout. By this time, the old leader realizes that it is not the leader anymore, steps down, and stops serving reads.

5.3.5 Correctness

Orca never returns non-monotonic states, i.e., a read from a client always returns at least the latest state that was previously read by any client. We now provide a proof sketch for how Orca ensures correctness under all scenarios.

First, when the current leader is functional, if a non-durable item (whose update-index is L) is read, Orca ensures that the state at least up to L is persisted on all the nodes in the active set before serving the read. Thus, reads performed at any node in the active set will return at least the latest state that was previously read (i.e., up to L). Followers not present in the active set may be lagging but reads are not allowed on them, preventing them from serving an older state. When a follower is added to the active set, Orca ensures that the follower contains state at least up to L; thus any subsequent reads on the added follower will return at least the latest state that was previously read, ensuring correctness. When the

leader removes a follower, Orca ensures that the follower marks itself out before the leader returns any data by committing it on the new reduced set, which prevents the follower from returning any older state.

When the current leader fails, Orca (similar to Cad) ensures that latest state that was read by clients survives into the new view, and thus future active sets. Let us suppose that the latest read has seen state up to index L. Because Orca persists the data on all the nodes in the active set and given that the active set contains at least a majority of nodes, at least one node in any majority will contain state up to L on its disk. Thus, only a candidate that has state at least up to L can get votes from a majority and become the leader. Thus, the latest state that was read by clients survives into the new view.

5.3.6 Implementation

We build Orca by implementing the above design of cross-client monotonic reads atop the Cad version of ZooKeeper (described in §4.4). Compared to the Cad implementation, the following additional changes were required. We modified the replication requests and responses in ZooKeeper to maintain the active-set lease. Instead of setting durable-index to the highest persisted-index among at least a majority as in Cad in Orca we set the durable-index as the maximum index that has been persisted and applied (to the state machine) by all nodes in the active set. We set the different timeouts as follows. We set the follower mark-out timeout to the same value as the heartbeat interval (100 ms in our implementation). We set the removal timeout to 500 ms. The leader adds a follower to the active set when the follower has caught up and responds to three consecutive requests promptly.

5.4 Evaluation

We now evaluate the performance of Orca against two versions of ZooKeeper: strong-ZK and weak-ZK. Strong-ZK is ZooKeeper with immediate durability (with batching), and with reads restricted to the leader; strong-ZK provides linearizability and thus cross-client monotonic reads. Weak-ZK replicates and persists writes asynchronously, and allows reads at all replicas; weak-ZK does not ensure cross-client monotonic reads. Orca uses the Cad durability layer and reads can be served by all replicas in the active set; we configure the active set to contain four replicas in our experiments. We use the same experimental setup as the one described in the previous chapter (§4.5). In our evaluation, we ask the following questions:

- What are the benefits of allowing reads at multiple nodes?
- How does Orca perform compared to weakly consistent ZooKeeper and strongly consistent ZooKeeper?
- What benefits does Orca offer in geo-replicated settings?
- Does Orca ensure cross-client monotonic reads in the presence of failures?

5.4.1 Read-only Micro-benchmark

We first demonstrate the benefit of allowing reads at many replicas using a read-only benchmark. Figure 5.8 plots the average latency against the read throughput for the three systems when varying the number of clients from 1 to 100. Strong-ZK restricts reads to the leader to provide strong guarantees, and so its throughput saturates after a point; with many concurrent clients, reads incur high latencies. Weak-ZK allows reads at many replicas and so can support many concurrent clients, leading to high throughput and low latency; however, the cost is weaker guarantees as we show

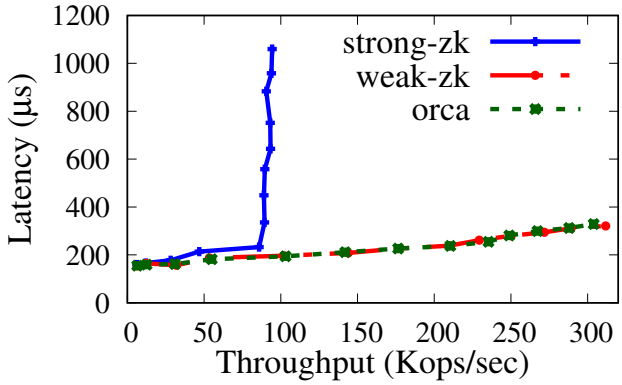


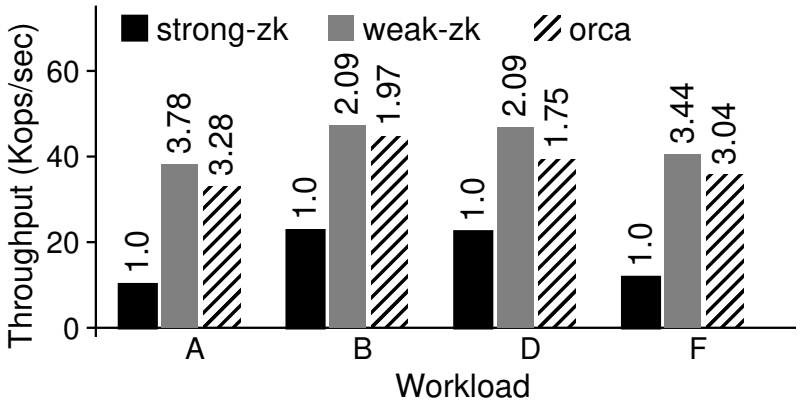
Figure 5.8: **ORCA Performance: Read-only Micro-benchmark.** *The figure plots the average latency against throughput by varying the number of clients for a read-only workload for the three systems.*

soon (§5.4.4). In contrast, ORCA provides strong guarantees while allowing reads at many replicas and thus achieving high throughput and low latency. The throughput of weak-ZK and ORCA could scale beyond 100 clients, but we do not show that in the graph.

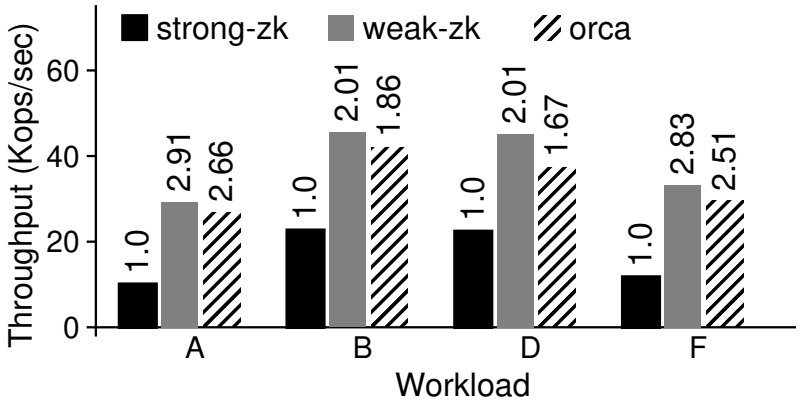
5.4.2 YCSB Macro-benchmarks

We now compare the performance of ORCA against weak-ZK and strong-ZK across different YCSB workloads. As we described earlier (§4.5), YCSB workloads are representative of applications such as session stores, photo-tagging, and user status updates. These workloads capture the read-write patterns exhibited by real-world applications; for example, workload D in YCSB has a latest access pattern where most reads access the recently written data items. Our goal in this experiment is to demonstrate the performance benefits of ORCA under these real-world workloads. We run this experiment with 10 clients. Figure 5.9 shows the results.

In Figure 5.9(a), weak-ZK and ORCA carry out both replication and persistence lazily; whereas, in 5.9(b), weak-ZK and ORCA replicate synchronously but persist to storage lazily, i.e., they issue *fsync*-s in the back-



(a) Baseline: async replication & persistence



(b) Baseline: async persistence

Figure 5.9: **Orca Performance.** *The figure compares the throughput of the three systems across different YCSB workloads. In (a), weak-ZK and* Orca *asynchronously replicate and persist; in (b), they replicate synchronously but persist data lazily. The number on top of each bar shows the performance normalized to that of strong-ZK.*

ground. As shown in Figure 5.9(a), Orca is notably faster than strong-ZK (3.04 – 3.28× for write-heavy workloads, and 1.75 – 1.97× for read-heavy workloads). Orca performs well due to two reasons. First, it avoids the cost of synchronous replication and persistence during writes. Second, it allows reads at many replicas, enabling better read throughput. Orca also closely approximates the performance of weak-ZK: Orca is only about 11% slower on an average. This reduction arises because reads that ac-

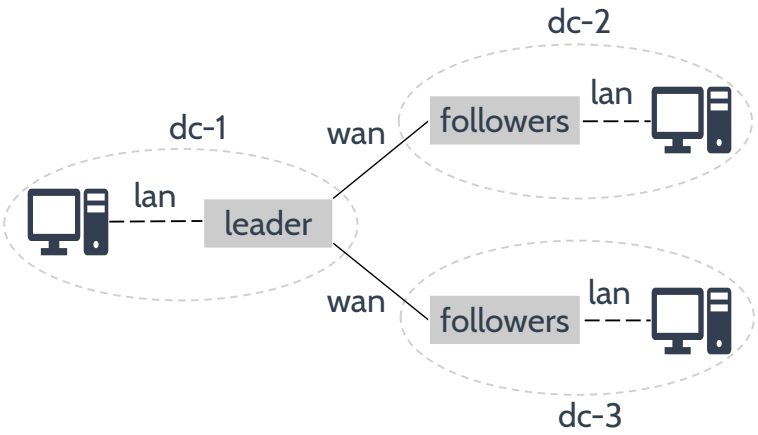


Figure 5.10: **Geo-distributed Experiment.** *The figure shows how the replicas and clients are located across multiple data centers in the geo-distributed experiment.*

cess non-durable items must persist data on all the nodes in the active set (in contrast to only a majority as done in CAD); further, reads at the followers that access non-durable data incur an additional round trip because they are redirected to the leader. Similar results and trends can be seen for the asynchronous-persistence baseline in Figure 5.9(b).

5.4.3 Performance in Geo-Replicated Settings

We now analyze the performance of ORCA in a geo-replicated setting by placing the replicas in three data centers (across the US), with no data center having a majority of replicas. The replicas across the data center are connected over WAN. We run the experiments with 24 clients, with roughly five clients near each replica. Figure 5.10 shows this setup. In weak-ZK and ORCA, reads are served at the closest replica; in strong-ZK, reads go only to the leader. In all three systems, writes are performed only at the leader.

Figure 5.11 shows the distribution of operation latencies across different workloads. We differentiate two kinds of requests: ones originating near the leader (the top row in the figure) and ones originating near the

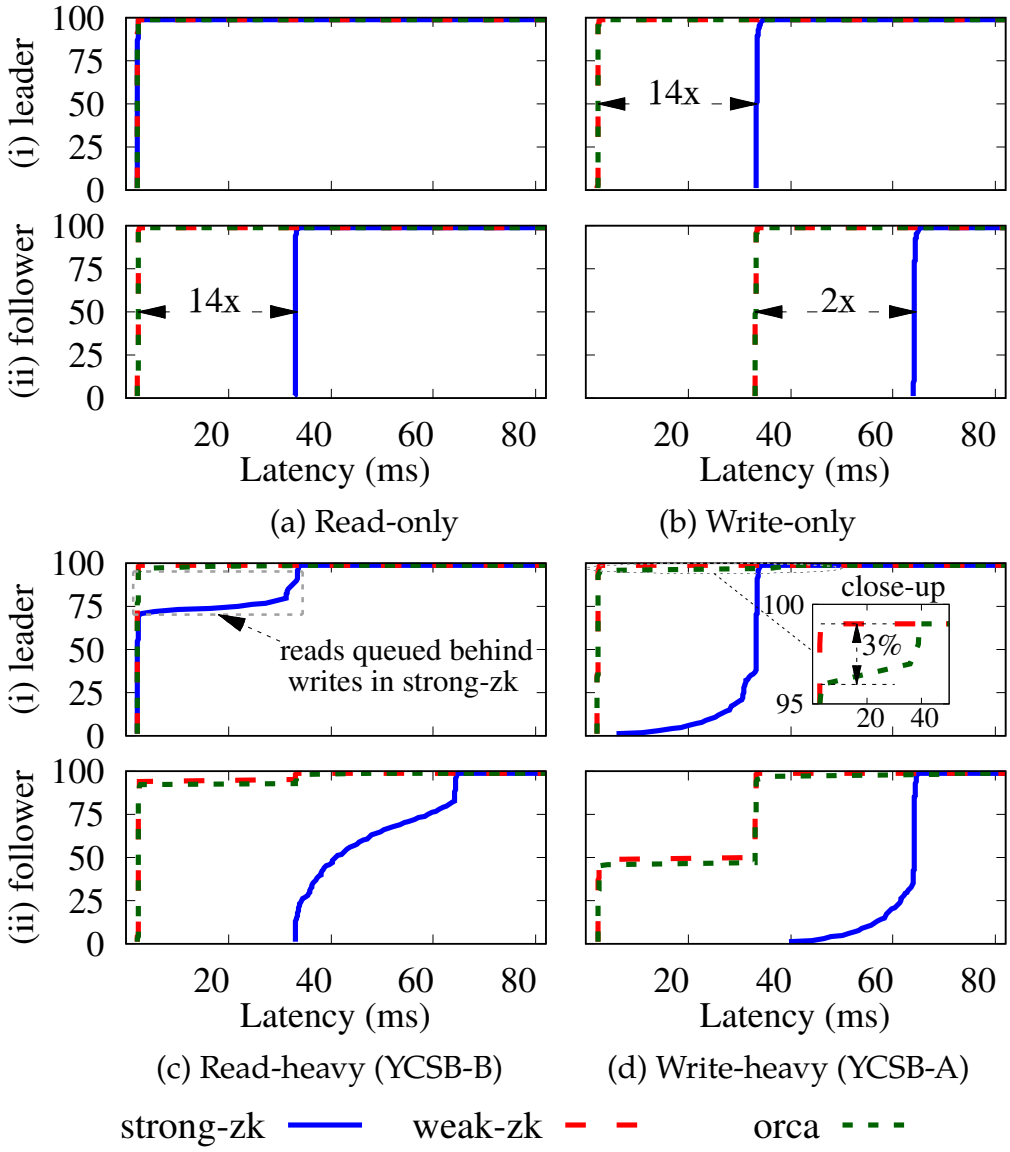


Figure 5.11: **Geo-distributed Latencies.** *The figure shows the distribution of operation latencies across different workloads in a geo-distributed setting. For each workload, (i) shows the distribution of latencies for operations originating near the leader; (ii) shows the same for requests originating near the followers. The ping latency between a client and its nearest replica is <2ms; the same between the client and a replica over WAN is ~35 ms.*

followers (the bottom row). As shown in Figure 5.11(a)(i), for a read-only workload, in all systems, reads originating near the leader are completed locally and thus experience low latencies (~2 ms). Requests originating near the followers, as shown in 5.11(a)(ii), incur one WAN RTT (~33 ms) to reach the leader in strong-ZK; in contrast, weak-ZK and Orca can serve such requests from the nearest replica and thus incur 14× lower latencies.

For a write-only workload, in strong-ZK, writes originating near the leader must incur one WAN RTT (to replicate to a majority) and disk writes, in addition to the one local RTT to reach the leader. In contrast, in weak-ZK and Orca, such updates can be satisfied after buffering them in the leader's memory, reducing latency by ~14×. Writes originating near the followers in strong-ZK incur two WAN RTTs (one to reach the leader and other for majority replication) and disk latencies; such requests, in contrast, can be completed in one WAN RTT in weak-ZK and Orca, reducing latency by ~2×.

Figure 5.11(c) and 5.11(d) show the results for workloads with a read-write mix. As shown, in strong-ZK, most operations incur high latencies; even reads originating near the leader sometimes experience high latencies because they are queued behind slow synchronous writes as shown in 5.11(c)(i). In contrast, most requests in Orca and weak-ZK can be completed locally and thus experience low latencies, except for writes originating near the followers that require one WAN RTT, an inherent cost in leader-based systems (e.g., 50% of operations in Figure 5.11(d)(ii)). Some requests in Orca incur higher latencies because they read recently modified data. However, only a small percentage of requests experience such higher latencies as shown in Figure 5.11(d)(i).

Orca performance summary. By avoiding the cost of synchronous replication and persistence during writes, and allowing reads at many replicas, Orca provides higher throughput (1.8 – 3.3×) and lower latency than strong-ZK. In the geo-distributed setting, Orca significantly reduces la-

tency (14×) for most operations by allowing reads at nearby replicas and hiding WAN latencies with asynchronous writes. ORCA also approximates the performance of weak-ZK. However, as we show next, ORCA does so while enabling strong consistency guarantees that weak-ZK cannot offer.

5.4.4 ORCA Consistency

We now check if ORCA's implementation correctly ensures cross-client monotonic reads in the presence of failures and also test the guarantees of weak-ZK and strong-ZK under failures. To do so, we used the crash-testing a framework described in the previous chapter (§4.5). The framework drives the cluster to different states by injecting crash and recovery events and message delays. At each intermediate cluster state of a crash sequence, we insert new items and perform reads on the non-delayed nodes. Then, we perform a read on the delayed node, triggering the node to return old data, thus exposing non-monotonic states. Every time we perform a read, we check whether the returned result is at least as latest as the result of any previous read. Using the framework, we generated 500 random sequences. We subject weak-ZK, strong-ZK, and ORCA to the generated sequences.

Table 5.1(a) shows results when weak-ZK and ORCA synchronously replicate but asynchronously persist. With weak-ZK, non-monotonic reads arise in 83% of sequences due to two reasons. First, read data is lost in many cases due to crash failures, exposing non-monotonic reads. Second, delayed followers obliviously serve old data after other nodes have served newer state. Strong-ZK, by using immediate durability and restricting reads to the leader, avoids non-monotonic reads in all cases. Note that while immediate durability can avoid non-monotonic reads caused due to data loss, it is not sufficient to guarantee cross-client monotonic reads. Specifically, as shown in the table, sync-ZK-all, a configuration that uses immediate durability but allows reads at all nodes, does not prevent lag-

System	Outcomes (%)	
	Correct	Non-monotonic
weak-ZK	17	83
strong-ZK	100	0
sync-ZK-all	63	37
ORCA	100	0

(a) Async persistence

System	Outcomes (%)	
	Correct	Non-monotonic
weak-ZK	4	96
strong-ZK	100	0
sync-ZK-all	63	37
ORCA	100	0

(b) Async replication & persistence

Table 5.1: **ORCA Correctness.** *The tables show how* ORCA *provides cross-client monotonic reads. In (a), weak-ZK and* ORCA *use asynchronous persistence; in (b), both replication and persistence are asynchronous.*

ging followers from serving older data, exposing non-monotonic states. In contrast to weak-ZK, ORCA does not return non-monotonic states. In most cases, a read performed on the non-delayed nodes persists the data on the delayed follower too, returning up-to-date data from the delayed follower. In a few cases (about 13%), the leader removed the follower from the active set (because the follower is experiencing delays). In such cases, the delayed follower rejects the read (because it is not in the active set); however, retrying after a while returns the latest data because the leader adds the follower back to the active set. Similar results can be seen in Table 5.1(b) when weak-ZK and ORCA asynchronously replicate and persist

writes.

5.5 Application Case Studies

We now show how the guarantees provided by ORCA can be useful in two application scenarios. The first one is a location-sharing application in which an user updates their location (e.g., a $\rightarrow$ b $\rightarrow$ c) and another user tracks the location. To provide meaningful semantics, the storage system must ensure monotonic states for the reader; otherwise, the reader might incorrectly see that the user went backwards. While systems that provide session-level guarantees can ensure this property within a session, they cannot do so across sessions (e.g., when the reader closes the application and re-opens, or when the reader disconnects and reconnects). Cross-client monotonic reads, on the other hand, provides this guarantee irrespective of sessions and failures.

We test this scenario by building a simple location-tracking application. A set of users constantly update their locations on the storage system, while another set of users constantly read those locations. We use the following workload mix: 50% update-location and 50% read-location. A set of 100 users connect to five application servers that run the location-tracking application. Each user connects to different servers over the lifetime of the experiment. We introduce delays between servers to create lagging nodes.

Table 5.2 shows result. As shown, weak-ZK exposes inconsistent (non-monotonic) locations in 13% of reads and consistent but old (stale) locations in 39% of reads. In contrast to weak-ZK, ORCA prevents non-monotonic locations, providing better semantics. Further, it also reduces staleness because of prior reads that make state durable. As expected, strong-ZK never exposes non-monotonic or old locations.

The second application is similar to Retwis, an open-source Twitter

Outcome(%)	**Location-tracking**			**Retwis**		
	weak-ZK	strong-ZK	Orca	weak-ZK	strong-ZK	Orca
Inconsistent	13	0	0	8	0	0
Consistent (old)	39	0	7	20	0	12
Consistent (latest)	48	100	93	72	100	88

Table 5.2: **Case Study: Location-tracking and Retwis.** *The table shows how applications can see inconsistent (non-monotonic), and consistent (old or latest) states with weak-ZK, strong-ZK, and* Orca.

clone [148]. Users can either post tweets or read their timeline (i.e., read tweets from users they follow). If the timeline is not monotonic, then users may see some posts that may disappear later from the timeline, providing confusing semantics [40]. Cross-client monotonic reads avoids this problem, providing stronger semantics for this application.

We run this experiment with 100 users where some users are popular (i.e., they have many followers). The workload in this application is read-dominated: most requests retrieve the timeline, while a few requests post new content. We thus use the following workload mix: 70% get-timeline and 30% posts, leading to a total of 95% reads and 5% writes for the storage system. The users connect to one of the five application servers with each user connecting to different servers at different points in time. To create nodes that are lagging, we introduce delays between the replicas. Non-monotonic reads can be exposed under two cases: first, when a user sees a post and it disappears later; second, when one user sees an older post than what other users have seen so far.

Results are similar to the previous case study. Weak-ZK returns non-monotonic and stale timelines in 8% and 20% of get-timeline operations,

respectively. ORCA completely avoids non-monotonic timelines and reduces staleness, providing better semantics for clients.

While we show that the guarantees provided are useful for applications we have not analyzed whether or not ORCA can simplify the recovery code employed by applications to handle inconsistent states. However, we believe that ORCA may potentially reduce the complexity of recovery code that handles stale and out-of-order data; specifically, such code now is freed from handling out-of-order data and may be less frequently invoked to handle stale version given that ORCA reduces staleness. Moreover, on systems built atop eventual durability, both writers and readers have to deal with data loss. With ORCA, only writers have to deal with data loss. For example, in the former, readers may have taken some action based on some data that they have read and they might later notice that is not true; therefore, the readers might have to perform some roll backs. This recovery code can be avoided with ORCA. We leave this study of how ORCA reduces developer complexity as an avenue for future work.

5.6 Summary and Conclusions

A major focus of distributed storage research and practice has been the consistency model a system provides. Most of this focus has been towards how a system behaves in the presence of concurrent operations from clients under different consistency models. However, only a little attention has been paid towards another important aspect: the behavior of the system under failures such as replica crashes and client crashes. In this thesis, we bridge this gap by analyzing how failures and data durability affect the consistency guarantees a system provides.

In this chapter, we introduced, *cross-client monotonic reads*, a new consistency model that provide stronger guarantees: it prevents out-of-order states even in the presence of failures and across client sessions. We de-

signed Orca, a design of cross-client monotonic reads atop Cad for leader-based majority systems. We introduced the concept of active sets to provide cross-client monotonicity while allowing reads at multiple replicas.

We implemented a prototype of Orca in ZooKeeper. We showed that Orca offers significantly higher throughput ($1.8 - 3.3\times$) compared to strongly consistent ZooKeeper (strong-ZK) and closely matches the performance of weakly consistent ZooKeeper (weak-ZK). Through a series of robustness test, we showed that Orca provides cross-client monotonic reads under hundreds of failure sequences; in contrast, weak-ZK returns non-monotonic states in many cases. We also demonstrated how the guarantees provided by Orca could be useful in two application scenarios.

6 Related Work

In this chapter, we discuss other pieces of work that are related to this dissertation. We start by discussing the various studies that motivate our analysis of distributed systems on how they react to storage faults (§6.1). We then describe efforts related to storage fault injection (§6.2) and analyzing distributed system reliability (§6.3). We then discuss how the new durability primitive and consistency property introduced in this dissertation compare with existing work (§6.4 and §6.5). We finally describe how CAD and ORCA are related to other efforts that aim to improve distributed system performance (§6.6).

6.1 Corruption and Errors in Storage Stack

Several past studies have analyzed storage errors and corruption in detail [24, 25, 97, 112, 155, 156]. Specifically, Bairavasundaram et al. [25] and Schroeder et al. [155] show the prevalence of latent sector errors in disks. Similarly, studies have also shown the prevalence of data corruption in the real world [24, 124]. Furthermore, studies have shown that cheap near-line disks are more prone to errors and corruption than enterprise-class devices [22]. Since large-scale deployments often tend to use cheap hardware (and build reliability into the software), problems such as disk errors and corruption are increasingly important in such deployments.

These prior studies motivated us to study the effects of such faults in distributed storage systems.

6.2 Storage Fault Injection

Given the prevalence of storage faults in hard disks and SSDs, prior work has studied the effect of these faults in storage systems. As we discuss in this section, most of the prior work has focused on *local* storage systems such as file systems (§6.2.1) and stand-alone databases (§6.2.2). While our work draws from both bodies of work, it is unique in its focus on testing behaviors of distributed systems to storage faults.

6.2.1 File-system Studies

Our work on distributed storage reliability analysis was inspired by prior studies that analyze how local file systems such as ext3, IBM JFS, ReiserFS, and ZFS react to storage faults [26, 130, 187]. These studies carefully inject disk faults just beneath the file system and observe how the file system reacts to the fault.

The injection methodology used in these studies is type-aware: faults are not injected at random; rather, they are injected into various specific on-disk structures of the file system in a targeted fashion. Type-aware fault injection helps in quickly exercising several file-system code paths compared to random fault injection. The fault-injection methodology in CORDS is similar in that it introduces faults in a targeted way into various application-level on-disk structures.

The results from the file-system studies show that some file systems (such as ext3) do not employ checksums for user data and simply propagate corruption to applications. The results also show that some file systems (such as ZFS) use checksums for user data and hence transform an underlying corruption into read errors. These results imply that applica-

tions running atop local file systems that desire to maintain data integrity have to handle such situations, motivating our study.

6.2.2 Studies on Layers Above the File System

A few studies [164, 186] have analyzed how applications running atop local file systems react to storage faults. For example, Subramanian et al. [164] study how the MySql database engine reacts to disk corruptions. Similarly, Zhang et al. [186] study how file synchronization services (such as Dropbox) react in the presence of local file-system corruption. However, none of these prior studies examined distributed storage systems that are central in today's data centers. We believe our work is the first to comprehensively examine the effects of storage faults across many distributed storage systems. Our study is unique in that while single-machine applications rarely have ways to recover from local storage faults, distributed systems have an opportunity to do so, which our work examines.

In our follow-on work [10][‡], we inject storage faults into RSM (replicated state machine) systems, a special class of distributed system. While the study presented in the first part of the thesis uses a simple failure model in which we inject only one storage fault on one node at a time, the follow-on work explores a more sophisticated failure model. It introduces storage faults on multiple nodes, introduces additional failures such as partitions, and explores scenarios such as the presence of lagging nodes. Therefore, the follow-on work uncovers more problems in some of the systems (such as LogCabin and ZooKeeper) we studied in this thesis. The follow-on work also proposes a solution for the crash-corruption entanglement problem we introduce in this thesis which is an essential piece to correctly recover from storage faults in RSM systems.

[‡]not a part of this dissertation

6.3 Analyzing Distributed System Reliability

Our work on analyzing the reliability of distributed systems to storage faults is related to prior research and efforts towards examining distributed system correctness. This class of work includes model checking, fault injection, and bug studies. In this section, we discuss each of these related efforts towards analyzing distributed system reliability. At a high level, while all these approaches find reliability problems, none of them focuses on storage faults like our work.

6.3.1 Model Checkers and Bug Finding Tools

Several model checkers have succeeded in uncovering bugs in distributed systems [67, 88, 183]. These model checkers directly check the correctness of the implementation and expose corner-case bugs by exercising different possible reordering of events; for instance, the checkers reorder network messages, inject crashes and reboots to find bugs. CORDS exposes bugs that cannot be discovered by model checkers; model checkers typically do not inject storage-related faults. Moreover, our targeted fault-injection framework can examine large storage systems faster than model checkers, which typically suffer from state-space explosion.

Similar to model checkers, tools such as Jepsen [83] that test distributed systems are complementary to CORDS. Jepsen tests the guarantees provided by a system as follows. The tool issues several concurrent operations to the system and introduces network partitions. It then heals the partition and checks whether or not the distributed system violates any guarantees. For example, it checks if writes were lost and if the system violated some consistency or isolation guarantee that it is supposed to provide. CORDS is different in that it introduces storage faults. We believe one could combine these tools to uncover other vulnerabilities that cannot be detected by either of these tools in isolation.

Our previous work [12], PACE, studies how file-system crash behaviors affect distributed systems. However, the faults introduced by PACE occur only upon a crash, unlike block corruption and errors introduced by CORDS. PACE also discovered some of the problems exposed by CORDS. For example, when a node crashes during an update, it results in a corruption because of the partially updated data. Such corruptions caused by crashes were undetected by Redis, and therefore, PACE also discovered the problem where Redis spreads corrupted data to many nodes. However, CORDS exposes additional problems not exposed by PACE because of the targeted injection of storage faults.

6.3.2 Generic Fault Injection

Our work is also related to efforts that inject faults into systems and test their robustness [28, 64, 157, 172]. Several efforts have built generic fault injectors for distributed systems [49, 69, 162]. Most of these fault-injection frameworks aim to inject various types of faults and also emphasize the portability of the framework to several platforms and systems. For example, Han et al., described Doctor [69], a comprehensive framework that can inject processor, memory, and communication faults. CORDS differs from generic fault injectors through its specific focus on storage faults.

6.3.3 Bug Studies

A few recent bug studies [66, 184] have given insights into common problems found in distributed systems. Gunawi et al. [66] perform a comprehensive study of over twenty thousand bug reports of systems such as HDFS, HBase, and Cassandra. Among the bugs they study, some issues are related to data integrity; for example, HDFS detects a corruption but does not perform recovery correctly in some cases. Yuan et al. [184] show that 34% of catastrophic failures in their study are due to unanticipated

error conditions. Our results also show that systems do not handle read and write errors well; this poor error handling leads to harmful global effects in many cases.

We believe that bug studies and fault injection studies are complementary to each other; while bug studies suggest constructing test cases by examining sequences of events that have led to bugs encountered in the wild, fault injection studies like ours concentrate on injecting one type of fault and uncovering new bugs and design flaws.

6.4 Durability Semantics

We now discuss durability models that are similar to CAD, introduced in Chapter 4. CAD's durability semantic has a similar flavor to that of a few local file systems that delay durability for performance. Xsyncfs [115] delays writes to disk until the written data is externalized; data is externalized when a response is printed on the output screen or when a message is sent via the network. By delaying durability, Xsyncfs realizes high performance similar to an asynchronous file system while providing strong guarantees similar to a synchronous file system. Similarly, file-system developers have proposed the `O_RSYNC` flag [76] that provides similar guarantees to CAD. Although not implemented by many kernels [76], when specified in `open`, this flag blocks `read` calls until the data being read has been persisted to the disk.

While CAD has similarities to these ideas, prior work resolves the tension between durability and performance in a much simpler single-node setting and within the file system. In contrast, to the best of our knowledge, our work is the first to do so in replicated storage systems and in the presence of complex failures (e.g., network partitions).

BarrierFS' `fbarrier` [181] and OptFS' `osync` [38] provide delayed durability semantics similar to CAD; however, unlike CAD, these file systems

do not guarantee that data read by applications will remain durable after crashes. Moreover, similar to Xsyncfs and `O_RSYNC`, these efforts focus on local file systems unlike CAD.

Prior research on building distributed systems such as Blizzard [100] and RAMCloud [121, 163] employ asynchronous durability for performance. For example, RAMCloud introduces a buffered logging approach to durability where operations are synchronously logged to the memory of replicas but not persisted to disks in the critical path. Hence, the system may lose data in the presence of failures. Similarly, Blizzard [100] also employs delayed durability and provides prefix writes (i.e., only the tail of the recent updates may be lost). CAD, similar to these systems, also may lose the tail of recent updates. However, unlike these systems, CAD guarantees the durability of data that has been read by clients. Consequently, while it is possible to build cross-client monotonic reads atop CAD, it is hard to realize it in prior systems. However, we believe the idea of read-triggered durability can be applied to systems such as Blizzard to improve its guarantees while maintaining high performance.

6.5 Cross-client Monotonic Reads

Several consistency models have been proposed in distributed systems literature [176]. These range from strong consistency property such as linearizability [70], intermediate causal [91] and session [169] consistency models, to weak models such as eventual consistency [52]. Our work focuses on the interaction of consistency and durability; specifically, we show how durability affects consistency. Lee et al., identify and describe the durability requirements to realize linearizability [87]. In contrast, we study in more detail what consistency levels can be realized upon different durability models; further, we also explore how to design a new durability primitive that enables strong consistency with high performance.

To the best of our knowledge, cross-client monotonic reads is provided only by linearizability [87, 119]. However, linearizable systems require immediate durability and most linearizable systems prevent reads at the followers. Orca offers this property without immediate durability while allowing reads at many nodes. Other models such as causal consistency [23, 91, 96] do not provide in-order cross-client consistency because their guarantees hold only within a single client session and not across client sessions.

Gaios [29] offers strong consistency while allowing reads from many replicas. Although Gaios distributes reads across replicas, requests are still bounced through the leader and thus incur an additional delay to reach the leader. The leader also requires one additional round trip to check if it is indeed the leader, increasing latency further. In contrast, Orca allows clients to directly read from the nearest replica, enabling both load distribution and low latency. Orca avoids the extra round trip (to verify leadership) by using leases.

CRAQ [167] offers strong consistency for systems that use chain replication while allowing reads from all nodes. However, CRAQ is optimized for read-mostly workloads. Moreover, while reads incur low latencies in chain replication, writes incur high latencies; the latency incurred is proportional to the number of replicas and is often higher than the write latency incurred in immediately durable leader-based majority systems. In contrast, Orca offers low-latency writes by using Cad.

Orca's use of leases to provide strong consistency is similar to how prior systems use leases for similar purposes. For example, early work on distributed file systems has used leases to maintain client-side cache consistency [63].

6.6 Improving Distributed System Performance

Several approaches to improving the performance of distributed systems have been proposed. These approaches improve performance by using client-side speculation [65, 114, 179], server-side speculation [79, 81], exploiting commutativity [109] and network ordering [89, 128], and using inconsistent replication [185]. However, these prior approaches do not focus on addressing the overheads of data durability, an important aspect in storage systems which our work on CAD addresses. Further, most of these approaches focus on state machine replication, while our work examines and applies new ideas to a broader class of replicated storage systems.

ORCA avoids durability overheads by separating consistency from freshness: reads can be stale but never out-of-order. LazyBase [40] applies a similar idea to analytical processing systems in which reads access only older versions that have been fully ingested and indexed. However, such an approach often returns staler results than a weakly consistent system. In contrast, ORCA never returns staler data than a weakly consistent system; further, ORCA reduces staleness compared to weak systems by persisting data on many nodes upon reads (as shown by our experiments).

SAUCR [11] reduces durability overheads in the common case but compromises on availability for strong durability in rare situations (e.g., in the presence of many simultaneous failures). ORCA makes the opposite tradeoff: it provides better availability but could lose a few recent updates upon failures. Moreover, the ideas proposed in SAUCR are applicable only within a data center. In contrast, ORCA provides benefits for both single data-center and geo-replicated settings.

7
Conclusions and Future Work

Distributed storage systems have a simple but important goal: to store critical data durably. However, this seemingly simple goal is hard to realize in the presence of failures. In this dissertation, we studied the durability of distributed storage systems in the presence of failures that arise while and after the system makes data durable. We also proposed efficient methods for achieving durability and stronger consistency in these systems.

To keep user data safe, a distributed system redundantly stores many copies of data. The common expectation is that if one of the replicas fail, the system can still recover the data from other copies. However, in the first part of our dissertation, we showed that most modern distributed storage systems do not effectively use redundancy as a source of recovery to recover from faults that arise in the local storage layer of the individual replicas. We showed that even a single corruption or an inaccessible block in one of the nodes could result in data loss, unavailability, and spread of corruption from a corrupted replica to other intact replicas.

Ensuring durability of data in the critical path of a client request is expensive. On a write, a distributed system must replicate the data to many servers and force the data to the storage devices; these operations result in significant performance degradation. Many practitioners, therefore, turn off such synchronous operations in the critical path, for performance.

However, such systems may lose data and provide weak guarantees when failures happen.

Therefore, in the second part of this dissertation, we presented solutions that offer both strong guarantees and excellent performance. We introduced *consistency-aware durability* (CAD), that shifts the point of durability from writes to reads. Delaying durability of writes provides high performance; however, ensuring that data is durable before it is read enables strong consistency even in the presence of failures. Finally, we showed how a strong consistency property called *cross-client monotonic reads* could be realized upon CAD. This new consistency property can be beneficial in geo-distributed settings and edge-computing scenarios and can be useful in many application scenarios.

In this chapter, we first summarize each part of this dissertation (§7.1) and present the various lessons we learned through the course of this dissertation work (§7.2). We then discuss some directions for future work (§7.3).

7.1 Summary

We now provide a summary of the three parts of this dissertation.

7.1.1 Storage Faults Analysis

Modern distributed storage systems depend upon local file systems to store and manage data. The storage devices underneath the local file system may sometimes return corrupted data on a read; at times, a block on the storage medium may become inaccessible. Therefore, in the first part of this thesis, we studied how modern distributed storage systems react to such storage faults and whether these faults affect the durability of data in these systems.

To this end, we built a fault-injection framework called CORDS that systematically injects storage faults into applications that run atop local file systems. Our fault model is simple: we injected only a single storage fault on one replica at a time. Given that there are multiple intact copies of data, the common expectation is that redundancy will enable recovery from local storage faults. For instance, if one of the copies of the data item in the system gets corrupted, users would expect that the corrupted data will be recoverable from the intact copies on other replicas and that the users never see the corrupted data.

However, out study revealed that the reality is different from this common expectation: redundancy does not imply fault tolerance in many systems we studied. In many cases, a single storage fault on one replica resulted in problematic outcomes such as data loss, silent user-visible corruption, unavailability, query failures, or sometimes even the spread of corrupted data to other intact replicas. We found that these outcomes arise due to some fundamental root causes in storage fault tolerance that are common to many distributed storage systems. First, faults were often undetected locally by replicas, leading to harmful effects. Second, even when systems reliably detected faults, in most cases, they simply crashed instead of using redundancy to recover from the fault. Third, many systems did not discern corruptions caused due to crashes from storage corruptions, resulting in many data-loss cases. Then, we found that local fault-handling behaviors and global distributed protocols interacted in unsafe manners, leading to propagation of corruption or data loss. Finally, redundancy was underutilized as a source of recovery from storage faults.

7.1.2 Consistency-aware Durability

In the second part of the thesis, we focused on how systems make data durable, which has strong implications on both consistency and perfor-

mance. We found that two durability models are popular and most existing systems use either one of them. With immediate durability, when a client performs an update, the writes are replicated and persisted on many nodes before acknowledging clients. With eventual durability, writes are only lazily replicated and persisted after buffering it on the memory of one or a few nodes. Immediate durability enables strong consistency but at a high cost: poor performance. In contrast, with eventual durability, high performance can be realized, but this model leads to weak semantics, exposing stale and out-of-order data to applications. However, many deployments prefer eventual durability for performance, settling for weaker guarantees.

To this end, we introduced consistency-aware durability or CAD that provides stronger guarantees than eventual durability without forgoing performance. The key idea behind CAD is to shift the point the system makes data durable from writes to reads. By committing writes asynchronously, CAD achieves high performance; however, unlike eventual durability, CAD guarantees the durability of data items (through synchronous operations, if necessary) before serving out reads, enabling stronger guarantees even in the presence of failures. In most real-world workloads, it is natural that reads do not immediately follow writes, enabling CAD to realize high performance.

We designed and implemented CAD for leader-based majority systems by modifying ZooKeeper. We showed that ZooKeeper with CAD performs significantly ($1.5\times - 3\times$) faster than immediately durable ZooKeeper. CAD closely matches the performance of eventually durable ZooKeeper for many workloads. However, as we showed, CAD provides better guarantees than eventually durable ZooKeeper and enables one to build stronger consistency.

7.1.3 Cross-client Monotonic Reads

In the last part of the thesis, we introduced cross-client monotonic reads, a new consistency model. Cross-client monotonicity guarantees that a read from a client will return a state that is at least as up-to-date as the state returned to a previous read from any client, irrespective of failures and across client sessions. Among the existing consistency models, these guarantees are only provided by linearizability; however, linearizability is expensive as it requires writes to be immediately durable. In contrast, we showed that CAD enables cross-client monotonicity to be realized with high performance.

We designed cross-client monotonicity upon the CAD version of ZooKeeper to build ORCA. In addition to using a consistency-aware durability layer, ORCA uses additional mechanisms to allow reads at many nodes (unlike many practical linearizable systems that restricts reads to the leader). Through rigorous experiments, we showed that ORCA provides strong consistency while approximating the performance of weakly consistent ZooKeeper. ORCA provides significantly higher throughput ($1.8\times$ – $3.3\times$) compared to strongly consistent ZooKeeper and notably reduces latency (by order of magnitude) in geo-distributed settings.

7.2 Lessons Learned

In this section, we present a list of general lessons we learned while working on this dissertation.

Even battle-tested systems can fall short in reliability measures. When we started working on the storage fault analysis project, we initially expected the systems we studied to handle storage faults reasonably well partly because most of them are widely used in practice. We did not expect to find serious vulnerabilities such as data loss and silent corruption.

Although we expected to find some implementation-level bugs that will be triggered due to code paths that are rarely tested, we did not expect to find fundamental problems that are common to many systems. However, through the course of the work, we realized that even battle-tested systems have reliability problems. More broadly, we believe while it is important to channel research and development efforts towards improving performance, it is even more critical to pay attention to reliability. However, our study in this dissertation suggests that this is not the case at least with respect to storage-fault tolerance in many modern distributed storage systems.

Systems lack uniform approaches to storage fault handling. In the first part of this thesis, we discovered several problems related to storage fault handling in distributed systems. The underlying cause for most of these problems is that there is no unified approach to handling storage faults across systems. A vast body of work [34, 77, 85, 117, 120] exists on how to tolerate fail-stop, fail-recover, and Byzantine failures in distributed systems. However, we learned that only scant attention has been paid to problems that arise at the local storage layer and that there is no widespread knowledge on how to build storage-fault-tolerant distributed systems. Our dissertation takes the initial but important step in this direction.

Decoupling of concerns and paying attention to lower layers can be beneficial. One of the contributions of the thesis is separating the concerns of durability and consistency and focusing on the underlying durability layer. This separation resulted in many benefits. First, it enabled the understanding of how the underlying durability layer affects the consistency guarantees that a system can provide. Next, it also allowed us to focus on failures: while consistency models mainly focus on how a system behaves in the presence of concurrent client operations, looking at the durability layer separately enabled us to focus on the system behavior in

the presence of failures. Finally, it also enabled us to rethink the durability layer to build a new durability primitive that enables both stronger consistency and high performance.

Building reliability-testing tools is important. We believe that it is important to build tools that test the reliability of existing systems. While recent research has taken strides in building new storage systems that are verified for correctness [36, 37, 158], we believe that building tools is still critical to improving the reliability of *existing systems*. For example, in Chapter 3, we built a fault-injection tool using which we discovered many previously unknown problems related to storage fault tolerance in existing distributed systems. The tool also helped expose some fundamental shortcomings in these systems in the way they handle storage faults. Fixing the problems we found can significantly improve the reliability of these systems. Overall, we believe finding and fixing reliability problems is a promising way to improve the reliability of existing systems.

Testing tools can be used to check not only existing systems and expose problems in them, but they can also be used to test the robustness of the solutions we build. For example, in Chapters 4 and 5, we developed a rigorous cluster crash-testing framework that introduces complex sequences of failure events such as node crashes, node recoveries, partitions, and delays. Putting our system through hundreds of failure sequences helped us identify and fix corner-case bugs in our implementation; it also exposed problems in the unmodified baselines.

7.3 Future Work

In this section, we outline directions in which our work could be extended in the future.

7.3.1 Storage Faults in Blockchain Systems

In this thesis, we focused on analyzing the durability of modern distributed storage systems that are part of a single administrative domain. One interesting avenue of future work is to explore the reliability of decentralized systems such as blockchains [111, 182] through systematic fault injection, specifically by introducing faults in the local storage layer of the nodes in the network.

Blockchains, in essence, are replicated storage systems that store data upon many commodity computers. Each node, in turn, uses a local storage stack to store a copy of the ledger. If blockchains are implemented poorly, an unsafe interaction with storage stack can result in catastrophic outcomes such as denial-of-service attacks, hard chain forks, and even double spending of money. Unsurprisingly, storage failures have caused severe problems in blockchains in the wild [2–5, 150]. Despite such devastating consequences, little has been done to systematically study how blockchain implementations tolerate faults that arise at the storage layer. Moreover, ledger-based systems are more prone to storage faults given their decentralized nature; many diverse, commodity computers with less reliable hardware and software participate in the network. Thus, we find it imperative to study and understand the resiliency of blockchains to storage faults in a principled manner.

7.3.2 CAD for Other Systems

In this thesis, we designed and implemented a new durability model (CAD) for distributed storage systems that have a single leader and require a majority of nodes to be available. However, the idea behind CAD is general and can be applied to other classes of systems that do not have a single leader or are available even when more than a majority of nodes have failed. These classes of systems include practical systems such as Cassan-

dra [13] and Riak [149], and systems built by prior research [23, 91, 96].

However, applying CAD to other systems is not straightforward and requires solving a few challenges. Most systems that do not have a single leader do not establish a single total ordering of operations. They might establish only a partial order such as ordering only causally-related operations; some systems may not even establish a causal order. In leader-based systems, checking for the durability of a data item is simple: CAD maintains a single durable index for the entire system that indicates the index of the latest update that is durable. Since all updates are ordered, the durability of a data item can be determined based on whether or not the index of the latest update that modified the data item is greater than the durable index. However, for systems that do not establish this total ordering, it is not sufficient to maintain a single durable index for the entire system. Instead, the system might need to maintain metadata for each item denoting whether it is durable or not.

Moreover, the guarantee of what items are durable on a read would differ. In our CAD implementation, when a non-durable item x is read, CAD makes the entire state (containing all updates up to the latest update that modified x) durable. For other classes of systems, the system might need to make only the updates to x or those updates that are causally related to x durable.

Another challenge is related to implementing cross-client monotonic reads on top of CAD for these highly available systems. Systems such as Cassandra are available for reads and writes even when only a single node is up. In such cases, to provide strong guarantees while maintaining high availability is challenging. For example, a node that is partitioned will not be part of the active set in ORCA; such a design may not be feasible with systems such as Cassandra. Maintaining multiple versions of data and serving only versions that have reached all the nodes may help address this problem. However, this design may increase staleness and thus

requires more careful thought.

Another type of system that future work could consider is distributed file systems. Distributed file systems such as HDFS [14] and Ceph [35] write to all replicas in the synchronous path of writes. In these systems, the idea of CAD can be applied; writes can be made asynchronous to improve performance. However, ensuring that writes are durable before serving reads can provide strong guarantees for clients.

7.3.3 Transactions upon CAD and ORCA

In this thesis, we focus on non-transactional storage systems such as key-value stores where each request updates or reads only a single data item. However, some replicated systems provide support for transactions where each transaction may access multiple data items. We believe it would be interesting to explore how to apply the ideas of CAD and ORCA for such systems. While CAD may not be able to support immediate durability of transactions, we believe that one can build strong isolation guarantees such as serializability on top of CAD. Serializability [6], while a strong isolation guarantee and often considered to be a gold-standard for databases, does not require transactions to be ordered in real-time. So, a transaction T_2 that begins in real-time after another transaction T_1 need not see the effects of operations performed as part of T_1. CAD and cross-client monotonic reads can enable one to build stronger guarantees by ensuring that if a transaction T_2 notices the effects of T_1, then all subsequent transactions will notice the effects of T_1 as well.

Another aspect that future work can build upon is studying the interaction between CAD and sharding. In this dissertation, we focused on a single shard that is replicated. However, as we discussed in §2.1, most distributed systems partition their data into multiple shards, and each shard is replicated. A distributed transaction might access data items across multiple shards. Therefore, while making data items durable in a shard,

the system might need to keep track of data items that need to be made durable on other shards. We believe that solving these challenges and applying the ideas behind CAD for transactional storage systems could offer performance benefits.

7.3.4 Caching on CAD and ORCA

One or more layers of cache in between the client and the storage system store data items so that future accesses can be faster and the overall load on the storage system can be reduced. However, caching often interacts with the consistency guarantees of the storage system. For example, consider that a replicated storage system provides the strongest guarantee (i.e., linearizability) and thus prevents stale reads. However, if a client reads data that is cached elsewhere, the client may notice stale data. As a result, the system can also expose out-of-order reads to clients. Similarly, even if the storage system provides cross-client monotonicity, caches can expose out-of-order states to clients.

Write caching also presents an interesting challenge for CAD. When a non-durable data item is cached, a read served by the cache cannot guarantee the durability of data in the storage system. However, such write caching is often not used in practice; for example, at Facebook [116] data is cached only on reads and not on writes; applications issue writes to the storage system directly and later invalidate the cache. Since data items are cached only on reads, guarantees of CAD can be ensured. However, if data is cached during writes, then one must co-design the caching layer and storage system; the cache has to interact with the storage system to learn if a data item is durable and only serve data items that are durable upon reads. Other reads can be treated as a cache miss and redirected to the storage system. We leave this co-design and the analysis of the interaction between caching and consistency as avenues for future work.

7.3.5 Active sets and Linearizability

In Chapter 5, we introduced the technique of active sets. Active sets use a lease-based mechanism to provide cross-client monotonicity while allowing reads at multiple nodes. However, the idea behind active sets is not limited to ORCA and can be implemented on systems that provide other guarantees such as linearizability. To ensure linearizability while serving reads from many nodes, the idea of active sets can be employed as follows. On a write, the request must be committed on all the nodes in the active set; reads should be served by only by the nodes in the active set. We leave this application of the active sets technique to linearizable systems as another avenue for future work.

7.4 Closing Words

Distributed storage systems are at the heart of applications running in today's modern data centers. They are responsible for storing and managing all the valuable data that we generate. It is critical that these systems keep user data safe. This dissertation takes initial but important steps towards examining and improving the guarantees provided by existing systems in two ways.

First, we examined if and how storage faults affect durability in modern distributed storage systems after the system has stored redundant copies of data. Our analysis of eight popular systems revealed that the seemingly obvious things that we take for granted in distributed systems such as redundancy would provide fault tolerance is not the reality in these systems. In most cases, these systems do not handle problems that arise at the storage layer. The storage layer offers a great abstraction, hiding complex details from applications (including distributed storage systems). However, for an application to be truly reliable, it must be aware of the problems that can arise at the layers below and handle them ap-

propriately. This line of thought also resonates with the classical end-to-end arguments in system design; although lower layers may implement some reliability mechanisms, it is the ultimate responsibility of higher-level software to ensure end-to-end reliability. However, the behavior of existing systems that we studied in this dissertation is in contrast with the end-to-end principles.

While finding and fixing problems in existing systems is one way to improving durability in existing systems, some systems provide weaker guarantees by design. For example, many current systems employ eventual durability and thus can arbitrarily lose data and enable only weaker consistency guarantees. In this dissertation, we also presented solutions to improve the guarantees of these systems without losing performance. A key to providing high performance and strong guarantees is to pay attention to what is observable to clients. Specifically, although the durability layer may lose some recent updates, it may not be acceptable to lose data that has been externalized to clients that have already read them. Based on this insight, CAD delays synchronous operations until the point the data is externalized to clients, thereby hiding the cost required to provide strong guarantees.

We believe the ideas presented in this dissertation can apply to systems that run in today's data centers. At the same time, we also believe that some of these ideas can prove useful with upcoming hardware and deployment trends. First, with storage devices packing more bits, their error rates are expected to rise. For instance, next-generation QLC devices are already known to have more errors if writes are frequent [32]. Thus, the problems that we found in the first part of this dissertation will be more prevalent and important in systems that use such devices. Second, edge and IoT platforms are rapidly changing the way servers are deployed; instead of hosting all compute and storage resources in a centralized data center, these resources are now closer to end-users and ap-

plications. The performance and durability tradeoffs we studied in the second part of this dissertation become even more important in storage systems that span such edge and cloud servers. We believe the durability and consistency model proposed in this thesis can prove useful in such settings.

Bibliography

[1] Cords Tool and Results. http://research.cs.wisc.edu/adsl/Software/cords/.

[2] CVE-2013-2293. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2293.

[3] CVE-2013-3220. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3220.

[4] LevelDB corrupted compressed block contents errors. https://github.com/bitcoin/bitcoin/issues/12690.

[5] Leveldb/Table: Corruption on Data-block. https://github.com/ethereum/go-ethereum/issues/2568.

[6] Abadi, Daniel and Freels, Matt. Serializability vs Strict Serializability: The Dirty Secret of Database Isolation Levels. https://fauna.com/blog/serializability-vs-strict-serializability-the-dirty-secret-of-database-isolation-levels.

[7] Marcos K. Aguilera and D. Terry. The Many Faces of Consistency. *IEEE Data Eng. Bull.*, 39:3–13, 2016.

[8] Ramnatthan Alagappan. *Protocol-and Situation-aware Distributed Storage Systems*. The University of Wisconsin-Madison, 2019.

[9] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Aws Albarghouthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond Storage APIs: Provable Semantics for Storage Stacks. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*, Kartause Ittingen, Switzerland, May 2015.

[10] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Protocol-Aware Recovery for Consensus-Based Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, February 2018.

[11] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[12] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

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

[14] Apache. Hadoop Distributed File System (HDFS). `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[15] Apache. Kakfa. http://kafka.apache.org/.

[16] Apache. ZooKeeper. https://zookeeper.apache.org/.

[17] Apache. ZooKeeper Configuration Parameters. https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html#sc_configuration.

[18] Apache. ZooKeeper Guarantees, Properties, and Definitions. https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_guaranteesPropertiesDefinitions.

[19] Apache. ZooKeeper Leader Activation. https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_leaderElection.

[20] Apache. ZooKeeper Overview. https://zookeeper.apache.org/doc/r3.5.1-alpha/zookeeperOver.html.

[21] Apache ZooKeeper. ZooKeeper Programmer's Guide - ZooKeeper Stat Structure. https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html#sc_zkStatStructure.

[22] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.0 edition, May 2015.

[23] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, New York, NY, June 2013.

[24] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca

Schroeder. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[25] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, CA, June 2007.

[26] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[27] Radu Banabic and George Candea. Fast Black-box Testing of System Recovery Code. In *Proceedings of the EuroSys Conference (EuroSys '12)*, Bern, Switzerland, April 2012.

[28] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4), April 1990.

[29] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[30] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for Data Centers. Technical report, Google, 2016.

[31] Randal C Burns, Robert M Rees, and Darrell DE Long. An Analytical Study of Opportunistic Lease Renewal. In *International Conference on Distributed Computing Systems (ICDCS '01)*, Phoenix, AZ, April 2001.

[32] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 521–526. IEEE, 2012.

[33] Michael Calore. Ma.gnolia Suffers Major Data Loss, Site Taken Offline. `https://www.wired.com/2009/01/magnolia-suffer/`.

[34] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[35] Ceph. Ceph File System. `https://ceph.io/`.

[36] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Verifying a High-performance Crash-safe File System Using a Tree Specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[37] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[38] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic

Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[39] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[40] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of the EuroSys Conference (EuroSys '12)*, Bern, Switzerland, April 2012.

[41] CockroachDB. CockroachDB. `https://www.cockroachlabs.com/`.

[42] CockroachDB. Disk corruptions and read/write error handling in CockroachDB. `https://forum.cockroachlabs.com/t/disk-corruptions-and-read-write-error-handling-in-cockroachdb/258`.

[43] CockroachDB. Resiliency to disk corruption and storage errors. `https://github.com/cockroachdb/cockroach/issues/7882`.

[44] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IA, June 2010.

[45] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ 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, Lind-

say Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 261–264, Hollywood, CA, October 2012.

[46] Data Center Knowledge. Ma.gnolia data is gone for good. `http://www.datacenterknowledge.com/archives/2009/02/19/magnolia-data-is-gone-for-good/`.

[47] Datastax. Netflix Cassandra Use Case. `http://www.datastax.com/resources/casestudies/netflix`.

[48] DataStax. Read Repair: Repair during Read Path. `http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesReadRepair.html`.

[49] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations. In *Proceedings of the 2nd International Computer Performance and Dependability Symposium (IPDS '96)*, 1996.

[50] Jeff Dean. Building Large-Scale Internet Services. `http://static.googleusercontent.com/media/research.google.com/en//people/jeff/SOCC2010-keynote-slides.pdf`.

[51] 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 the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, October 2007.

[52] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, August 1987.

[53] etcd. etcd. `https://coreos.com/etcd`.

[54] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, Salt Lake City, Utah, 2012.

[55] Flavio Junqueira. Transaction Logs and Snapshots. `https://mail-archives.apache.org/mod_mbox/zookeeper-user/201504.mbox/%3CDA045626-54A4-4F8A-96C0-69DA574D9807@yahoo.com%3E`.

[56] Daniel Fryer, Dai Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the Integrity of Transactional Mechanisms. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, CA, February 2014.

[57] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[58] FUSE. Linux FUSE (Filesystem in Userspace) interface. `https://github.com/libfuse/libfuse`.

[59] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017.

[60] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Strong and Efficient Consistency with Consistency-aware Durability. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020.

[61] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, April 2018.

[62] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[63] Cary G. Gray and David Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, Litchfield Park, Arizona, December 1989.

[64] Weining Gu, Z. Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux Kernel Behavior Under Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, San Francisco, CA, June 2003.

[65] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[66] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, November 2014.

[67] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[68] James R Hamilton et al. On Designing and Deploying Internet-Scale Services. In *Proceedings of the 21st Annual Large Installation System Administration Conference (LISA '07)*, Dallas, Texas, November 2007.

[69] Seungjae Han, Kang G Shin, and Harold A Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS '95)*, 1995.

[70] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.

[71] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Sys-

tems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, MA, June 2010.

[72] Henrik Ingo and Aishwarya Ganesan. Discussion with Henrik Ingo. https://www.openlife.cc/comment/662091#comment-662091.

[73] James Myers. Data Integrity in Solid State Drives. http://intel.ly/2cF0dTT.

[74] Jay Kreps. Using forceSync=no in Zookeeper. https://twitter.com/jaykreps/status/363720100332843008.

[75] Jerome Verstrynge. Timestamps in Cassandra. http://docs.oracle.com/cd/B12037_01/server.101/b10726/apphard.htm.

[76] Jonathan Corbet. O_*SYNC. https://lwn.net/Articles/350219/.

[77] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[78] Kafka. Data corruption or EIO leads to data loss. https://issues.apache.org/jira/browse/KAFKA-4009.

[79] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All About Eve: Execute-verify Replication for Multi-core Servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, October 2012.

[80] Karthik Ranganathan. Low Latency Reads in Geo-Distributed SQL with Raft Leader Leases. https://blog.yugabyte.com/low-

latency-reads-in-geo-distributed-sql-with-raft-leader-leases/.

[81] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[82] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrisha Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Cambridge, MA, November 2000.

[83] Kyle Kingsbury. Jepsen. http://jepsen.io/.

[84] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28*, 9:690–691, September 1979.

[85] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[86] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[87] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[88] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[89] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[90] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT CSAIL, 2012.

[91] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[92] LogCabin. LogCabin. `https://github.com/logcabin/logcabin`.

[93] LogCabin. Reaction to disk errors and corruptions. `https://groups.google.com/forum/#!topic/logcabin-dev/wqNcdj0IHe4`.

[94] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[95] Mark Adler. Adler32 Collisions. http://stackoverflow.com/questions/13455067/horrific-collisions-of-adler32-hash.

[96] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, March 2017.

[97] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*, Portland, Oregon, June 2015.

[98] Ningfang Mi, A. Riska, E. Smirni, and E. Riedel. Enhancing Data Availability in Disk Drives through Background Activities. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[99] Michael Rubin. Google moves from ext2 to ext4. http://lists.openwall.net/linux-ext4/2010/01/04/8.

[100] James Mickens, Edmund B. Nightingale, Jeremy Elson, Krishna Nareddy, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, and Osama Khan. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, April 2014.

[101] MongoDB. MongoDB. https://www.mongodb.org/.

[102] MongoDB. MongoDB at eBay. https://www.mongodb.com/presentations/mongodb-ebay.

[103] MongoDB. MongoDB Read Preference. https://docs.mongodb.com/manual/core/read-preference/.

[104] MongoDB. MongoDB Replication. https://docs.mongodb.org/manual/replication/.

[105] MongoDB. MongoDB WiredTiger. https://docs.mongodb.org/manual/core/wiredtiger/.

[106] MongoDB. MongoDB YouGov. http://bit.ly/2GgGyeX.

[107] MongoDB. Non-Blocking Secondary Reads. https://www.mongodb.com/blog/post/mongodb-40-nonblocking-secondary-reads.

[108] MongoDB. Read Concern Linearizable. https://docs.mongodb.com/manual/reference/read-concern-linearizable/.

[109] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[110] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. Toward Session Consistency for the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, Boston, MA, July 2018.

[111] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[112] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu,

Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, Haifa, Israel, June 2016.

[113] Netflix. Cassandra at Netflix. `http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html`.

[114] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative Execution in a Distributed File System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005.

[115] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[116] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[117] 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*, ON, Canada, August 1988.

[118] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[119] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[120] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.

[121] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[122] Oracle. Fusion-IO Data Integrity. `https://blogs.oracle.com/linux/entry/fusion_io_showcases_data_integrity`.

[123] Oracle. Preventing Data Corruptions with HARD. `http://docs.oracle.com/cd/B12037_01/server.101/b10726/apphard.htm`.

[124] Bernd Panzer-Steindel. Data Integrity. *CERN/IT*, 2007.

[125] Parsely Inc. Streamparse: Configuring Zookeeper with forceSync = no. `https://github.com/Parsely/streamparse/issues/168`.

[126] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, et al. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[127] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[128] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, March 2015.

[129] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *The Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, Yokohama, Japan, June 2005.

[130] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005.

[131] Rahul Bhartia. MongoDB on AWS Guidelines and Best Practices. `http://media.amazonwebservices.com/AWS_NoSQL_MongoDB.pdf`.

[132] David Ratner, Peter Reiher, Gerald J Popek, and Geoffrey H Kuenning. Replication Requirements in Mobile Environments. *Mobile Networks and Applications*, 6(6):525–533, 2001.

[133] Redis. Instagram Architecture. http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html.

[134] Redis. Redis. http://redis.io/.

[135] Redis. Redis at Flickr. http://code.flickr.net/2014/07/31/redis-sentinel-at-flickr/.

[136] Redis. Redis Persistence. https://redis.io/topics/persistence.

[137] Redis. Redis Replication. http://redis.io/topics/replication.

[138] Redis. Redis Sentinel Documentation. https://redis.io/topics/sentinel.

[139] Redis. Redis WAIT. https://redis.io/commands/wait.

[140] Redis. Scaling Reads. https://redislabs.com/ebook/part-3-next-steps/chapter-10-scaling-redis/10-1-scaling-reads/.

[141] Redis. Silent data corruption in Redis. https://github.com/antirez/redis/issues/3730.

[142] RethinkDB. Integrity of read results. https://github.com/rethinkdb/rethinkdb/issues/5925.

[143] RethinkDB. RethinkDB. https://www.rethinkdb.com/.

[144] RethinkDB. RethinkDB Data Storage. https://www.rethinkdb.com/docs/architecture/#data-storage.

[145] RethinkDB. RethinkDB Doc Issues. https://github.com/rethinkdb/docs/issues/1167.

[146] RethinkDB. RethinkDB Faq. https://www.rethinkdb.com/faq/.

[147] RethinkDB. Silent data loss on metablock corruptions. https://github.com/rethinkdb/rethinkdb/issues/6034.

[148] Retwis. Retwis. https://github.com/antirez/retwis.

[149] Riak. Riak KV. https://riak.com//.

[150] Robert Escriva. Claiming Bitcoin's Bug Bounty. http://hackingdistributed.com/2013/11/27/bitcoin-leveldb/.

[151] Robert Harris. Data corruption is worse than you know. http://www.zdnet.com/article/data-corruption-is-worse-than-you-know/.

[152] Ron Kuris. Cassandra From tarball to production. http://www.slideshare.net/planetcassandra/cassandra-from-tarball-to-production-2.

[153] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[154] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4), 1984.

[155] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, CA, February 2010.

[156] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[157] D.P. Siewiorek, J.J. Hudak, B.H. Suh, and Z.Z. Segal. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, June 1993.

[158] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[159] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring Data Integrity in Storage: Techniques and Applications. In *The 1st International Workshop on Storage Security and Survivability (StorageSS '05)*, FairFax County, Virginia, November 2005.

[160] David Smith. The Cost of Lost Data. `https://gbr.pepperdine.edu/2010/08/the-cost-of-lost-data/`.

[161] Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Douglas B. Terry. Dealing with Server Corruption in Weakly Consistent Replicated Data Systems. *Wirel. Netw.*, 5(5), October 1999.

[162] David T. Stott, Benjamin Floering, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proceedings of the 4th International Computer Performance and Dependability Symposium (IPDS '00)*, Chicago, IL, 2000.

[163] Ryan Scott Stutsman. *Durability and Crash Recovery in Distributed In-Memory Storage Systems*. PhD thesis, Stanford University, 2013.

[164] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S Gunawi, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jeffrey F Naughton. Impact of Disk Corruption on Open-Source DBMS. In *Proceedings of the 26th International Conference on Data Engineering (ICDE '10)*, Long Beach, CA, March 2010.

[165] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[166] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. Who's Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy. In *Proceedings of the USENIX Annual Technical Conference (USENIX '19)*, Renton, WA, July 2019.

[167] Jeff Terrace and Michael J Freedman. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, CA, June 2009.

[168] 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 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, CO, December 1995.

[169] Doug Terry. Replicated Data Consistency Explained Through Baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[170] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session Guarantees for

Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems (PDIS '94)*, Autin, TX, September 1994.

[171] Theodore Ts'o. What to do when the journal checksum is incorrect. `https://lwn.net/Articles/284038/`.

[172] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems (MMB '95)*, London, UK, September 1995.

[173] Twitter. Kafka at Twitter. `https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time`.

[174] Uber. The Uber Engineering Tech Stack, Part I: The Foundation. `https://eng.uber.com/tech-stack-part-one/`.

[175] Uber. The Uber Engineering Tech Stack, Part II: The Edge And Beyond. `https://eng.uber.com/tech-stack-part-two/`.

[176] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, June 2016.

[177] Voldemort. Project Voldemort. `http://www.project-voldemort.com/voldemort/`.

[178] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, April 2013.

[179] Benjamin Wester, James Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines through Client Speculation. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, April 2009.

[180] Martyn Williams. Microsoft loses Sidekick users' personal data. `https://www.infoworld.com/article/2629952/microsoft-loses-sidekick-users--personal-data.html`.

[181] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, February 2018.

[182] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger.

[183] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, April 2009.

[184] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[185] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[186] 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 Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, CA, February 2014.

[187] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, CA, February 2010.

[188] ZooKeeper. Cluster unavailable on space and write errors. `https://issues.apache.org/jira/browse/ZOOKEEPER-2495`.

[189] ZooKeeper. Crash on detecting a corruption. `http://mail-archives.apache.org/mod_mbox/zookeeper-dev/201701.mbox/browser`.

[190] ZooKeeper. Zookeeper service becomes unavailable when leader fails to write transaction log. `https://issues.apache.org/jira/browse/ZOOKEEPER-2247`.