Protocol- and Situation-Aware Distributed Storage Systems

By

Ramnatthan Alagappan

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: July 8th, 2019

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
Michael M. Swift, Professor, Computer Sciences
Miguel Castro, Principal Researcher, Microsoft Research
Aws Albarghouthi, Assistant Professor, Computer Sciences
Kassem Fawaz, Assistant Professor, ECE

All Rights Reserved

© Copyright by Ramnatthan Alagappan 2019

Dedicated to my mom. Without her perseverance and grit, this would not have been possible.

Acknowledgments

This thesis would not have been possible without the guidance and support of many people whom I would like to thank.

First and foremost, I would like to express my deepest gratitude to my advisors Andrea and Remzi. I absolutely loved the balance of freedom they offered, and their involvement and willingness to help me in research. They gave me enough freedom to choose my own projects, collaborators, and deadlines. At the same time, they showed a great deal of commitment to my research, hardly missing our weekly one-on-one meetings. Their deep technical expertise, clear thinking, and great sense of humor made these meetings enjoyable, and I learned a whole lot. If I become a professor or a researcher, I need to look no further for ideal role models; I would strive to emulate Andrea and Remzi. I would consider myself successful if I achieve half of what they have accomplished individually. I honestly could not have asked for better advisors. Thank you, Andrea and Remzi!

Although Andrea and Remzi co-advise all students, they have very unique styles, and I have benefited immensely by learning from both. Andrea is a student-first professor: she puts the student's interests ahead of everything. She always encouraged me to choose problems that *I* was interested in. I cannot express how deeply Andrea has influenced my thoughts on how to organize ideas in a paper. Her prompt and incredibly

detailed feedback on every draft that I wrote during the last six years has helped me better organize and present complex ideas. Andrea's attention to detail is remarkable. Several times throughout my Ph.D., I have wondered how she catches the slightest of deviations in the data and tries to understand the underlying cause. I consider myself lucky to have worked with her.

Remzi has been a constant inspiration for both research and life. One characteristic of Remzi that has left me amazed is how one can be a top-notch researcher and have so much fun at the same time. Remzi has an uncanny talent for conveying complex ideas effectively and eloquently. Specifically, his advice on how to write lucid introductions to papers has always astonished me. Remzi has helped simplify my complicated and sometimes muddled ideas. On several occasions, he listened to my ideas, and conveyed them back but in a much more appealing and exciting way. Every talk I gave during my Ph.D. was greatly influenced by Remzi's advice. I have taken these pieces of advice to my heart and will apply them for every talk I will give. Remzi also caringly asked about my well-being and family periodically. I am greatly indebted to Remzi for what he has done for me.

I would also like to extend my gratitude to my committee members – Michael Swift, Miguel Castro, Aws Albarghouthi, and Kassem Fawaz. I took CS-739 (distributed systems) with Mike and was amazed by his breadth and depth of knowledge in systems. My first paper (presented in Chapter 5 of this thesis) started as a course project in this class. Mike's comments on my work have significantly improved the content and the presentation of this dissertation. I met Miguel at OSDI '18 for a brief time but was convinced that my work can benefit substantially from his feedback. So, I asked him (somewhat nervously) if he could serve on my committee. But, Miguel was very enthusiastic, and his challenging questions and comments on my work have significantly improved this dissertation.

Aws showed a keen interest in my work and was always available to help. I thank Kassem for his intriguing philosophical questions about my work.

Apart from the above, many others have helped me indirectly during my Ph.D. I would like to thank the CS staff on the fifth floor for simplifying many administrative tasks for graduate students. I would also like to extend my thanks to CSL and CloudLab teams. Most (if not all) of the experiments presented in this dissertation were conducted on the CloudLab infrastructure.

I was very fortunate to work with a set of smart and hardworking colleagues in our group: Samer Al-Kiswany, Leo Prasath Arulraj, Vijay Chidambaram, Tyler Harter, Jun He, Sudarsun Kannan, Jing Liu, Lanyue Lu, Yuvraj Patel, Neil Perry, Thanumalayan (Thanu) Sankaranarayana Pillai, Anthony Rebello, Zev Weiss, Kan Wu, Yien Xu, Leon Yang, Suli Yang, and Dennis Zhou. I would like to thank Vijay Chidambaram for his guidance during my initial years. Thanu has given me many pieces of valuable advice over the years and continues to do so. I enjoyed working with him on several projects. I also thank Sudarsun for his encouragement and advice.

I would also like to thank my friends in Madison: Sourav Das accompanied me in many courses during my initial years. Thanumalayan Sankaranarayana Pillai and Sanketh Nalli gave company during hard paper deadlines. Hanging out with Harshad Deshmukh, Ira Deshmukh, Adalbert Gerald, Supriya Hirurkar, Rogers Jeffrey, Kaviya Lakshmipathy, and Meenakshi Syamkumar helped me stay happy. Phone calls with Bharath Ramakrishnan reminded me to have fun.

I am grateful to my brothers Lex and Thiru for taking care of many filial responsibilities while I was in graduate school. I cannot thank my elder brother Lex enough for his belief in me and my abilities. I will never forget the time he left the beautiful summer in New Zealand to visit me during my first winter in Wisconsin. I thank my younger brother Thiru for always being there for me. I appreciate him for being more responsible

while I was away. I also thank Abi, Dhruv, and Aarav for their support. Video calls with Dhruv and Aarav always made me cheerful. I would also like to thank my parents-in-law and Akshaya for their constant support and encouragement.

This Ph.D. would not have been possible without my mom. My mom worked very hard to put her sons through school; without her efforts, I might not have even attended college. At every stage of life, she encouraged me to strive for excellence no matter how hard the situation was (like she does to this day in her life). Her courage and perseverance have a profound influence on my life. I dedicate this Ph.D. to her.

Finally, I do not have enough words to thank Aishwarya. Few people are fortunate to have an inspiring colleague, a trustworthy friend, or a great life partner. I am lucky to have a single person play all these roles in my life. Aishwarya, while being emotionally very supportive, has continuously challenged me throughout my Ph.D. by constructively criticizing my ideas. In research, she has set a very high bar for designing the right set of experiments and framing compelling answers to hard questions. I am happy that I worked with her on many projects; in fact, she is a co-author on all the papers that constitute this dissertation. In life, Aishwarya has taught me how to stay calm and think clearly amidst chaos. I look forward to many years of work and life with her.

Contents

Acknowledgments **ii**

Contents **vi**

List of Tables **xi**

List of Figures **xiii**

Abstract **xix**

1 Introduction **1**
- 1.1 Protocol-Aware Recovery from Storage Faults 3
- 1.2 Situation-Aware Updates and Crash Recovery 5
- 1.3 File-system Crash Behaviors in Distributed Systems 8
- 1.4 Contributions 12
- 1.5 Overview 14

2 Background **16**
- 2.1 Modern Distributed Storage Systems 16
- 2.2 Replicated State Machine Systems 21
  - 2.2.1 The Path of an Update Request 22
  - 2.2.2 Persistent Structures 23
  - 2.2.3 Handling Failures 24

2.3 Failure Models 24
2.3.1 Storage Faults 24
2.3.2 Crash Failures 26
2.3.3 File-system Crash Behaviors 28
2.4 Summary 29

3 Recovering from Storage Faults via Protocol-Awareness 30
3.1 Analysis of Existing Approaches 31
3.1.1 Storage Faults in Distributed Systems 31
3.1.2 RSM-based Storage Systems 32
3.1.3 Analysis Methodology 33
3.1.4 RSM Recovery Taxonomy 33
3.1.5 Summary: The Need for Protocol-Awareness 39
3.2 Corruption-Tolerant Replication 40
3.2.1 RSM Protocol-Level Attributes 42
3.2.2 Fault Model 44
3.2.3 Safety and Availability Guarantees 45
3.2.4 CTRL Local Storage Layer 45
3.2.5 CTRL Distributed Log Recovery 50
3.2.6 CTRL Distributed Snapshot Recovery 56
3.2.7 CTRL Summary 61
3.3 Implementation 62
3.3.1 Local Storage Layer 63
3.3.2 Distributed Recovery 64
3.4 Evaluation 65
3.4.1 Correctness 65
3.4.2 Performance 71
3.5 Summary and Conclusions 74

4 Situation-Aware Updates and Crash Recovery 76
4.1 Existing Approaches to Replication 77

4.1.1 Disk-Durable Protocols 77
4.1.2 Memory-Durable Protocols 79
4.1.3 Failure Patterns in Data Centers 83
4.1.4 Non-Reactiveness and Static Nature 85
4.1.5 Summary: The Need for a Situation-Aware Approach 86
4.2 Situation-Aware Updates and Recovery 87
4.2.1 Failure Model 89
4.2.2 Guarantees 89
4.2.3 SAUCR Modes 90
4.2.4 Failure Reaction 92
4.2.5 Enabling Safe Mode-Aware Recovery 93
4.2.6 Crash Recovery 95
4.2.7 Summary and Guarantees 100
4.3 Implementation 102
4.4 Evaluation 103
4.4.1 Durability and Availability 103
4.4.2 Performance 108
4.4.3 Heartbeat Interval vs. Performance 111
4.4.4 Correlated Failure Reaction 113
4.5 Discussion 114
4.6 Summary and Conclusions 116

5 File-system Crash Behaviors in Distributed Systems 118
5.1 Studying the Effects of File-system Crash Behaviors 119
5.1.1 Failure Model 119
5.1.2 Distributed Crash States 122
5.2 Protocol-Aware Crash Explorer 124
5.2.1 Design and Implementation Overview 125
5.2.2 Crash States 125
5.2.3 Protocol-Aware Exploration 131
5.2.4 Limitations and Caveats 138

5.3 Vulnerabilities Study 139
5.3.1 Workloads and Checkers 140
5.3.2 Vulnerability Accounting 142
5.3.3 Example Protocols and Vulnerabilities 145
5.3.4 Patterns in File-system Requirements 147
5.3.5 Vulnerability Consequences 150
5.3.6 Impact on Real File Systems 151
5.3.7 Confirmation of Problems Found 151
5.3.8 Discussion 152
5.4 Solving the Problems found by PACE 154
5.4.1 Local Hardening 154
5.4.2 Handling Using Distributed Redundancy 156
5.5 Summary and Conclusions 156

6 Related Work 158
6.1 Studies on Storage Faults 158
6.1.1 Prevalence of Storage Faults 158
6.1.2 File-system Behaviors to Storage Faults 159
6.1.3 Application Behaviors to Storage Faults 159
6.1.4 Distributed-system Reactions to Storage Faults . . . 160
6.2 Approaches to Handling Storage Faults 160
6.2.1 More Reliable Local Storage 161
6.2.2 Handling at the Distributed Layer 161
6.3 SAUCR Techniques 164
6.3.1 Failure Detection and Reaction 164
6.3.2 Situation-Aware Updates 165
6.3.3 Situation-Aware Recovery 165
6.3.4 Performance Optimizations in RSM systems 166
6.4 Testing Distributed Systems 167
6.5 Crash Vulnerabilities in Single-node Applications 169

7 Conclusions and Future Work **170**

7.1 Summary . . . 170

7.1.1 Storage Faults - Analysis and Solution . . . 171

7.1.2 Crash Resiliency and Performance - Analysis and Solution . . . 171

7.1.3 File-system Crash Behaviors in Distributed Systems 173

7.2 Lessons Learned . . . 173

7.2.1 The Importance of Measuring and then Building . . 174

7.2.2 Perspectives on Working on Widely Used Systems . 175

7.2.3 The Importance of Paying Careful Attention to Failures . . . 176

7.3 Future Work . . . 177

7.3.1 Applying PAR to Other Systems . . . 177

7.3.2 Minimizing Software Overheads on Fast Storage . . 178

7.3.3 Reasoning about Partial Failures . . . 179

7.3.4 Crash Consistency in Distributed Transactions . . . 180

7.3.5 Studying Interdependency Faults . . . 180

7.4 Closing Words . . . 181

Bibliography **183**

Appendices **217**

A Proof of Impossibility of Last-Entry Disentanglement **217**

List of Tables

3.1 **Recovery Taxonomy.** *The table shows how different approaches behave in Figure 3.1 scenarios. While all approaches are unsafe or unavailable,* CTRL *ensures safety and high availability.* 35
3.2 **Storage Fault Model.** *The table shows storage faults included in our model and possible causes that lead to a fault outcome.* 43
3.3 **Techniques Summary.** *The table shows a summary of techniques employed by* CTRL*'s storage layer and distributed recovery.* 61
3.4 **Targeted Corruptions.** *The table shows results for targeted corruptions in log; we trigger two policies (truncate and crash) in the original systems. Recovery is possible when at least one intact copy exists; recovery is not possible when no intact copy exists.* 66
3.5 **Log Recovery.** *(a) shows results for random block corruptions and errors in the log. (b) shows results for random corruptions in the log with crashed and lagging nodes.* 68
3.6 **Snapshot and FS Metadata Faults.** *(a) and (b) show how* CTRL *recovers from snapshot and FS metadata faults, respectively.* 70

4.1 **Disk Durability Overheads.** *The table shows the overheads of disk durability. The experimental setup is detailed in §4.4.2.* 78

4.2 **Durability and Availability.** *The table shows the durability and availability of memory-durable ZK (ZK-mem), VR (VR-ideal), disk-durable ZK (ZK-disk), and* SAUCR*. !min-rec denotes that only less than a bare minority are in recovered state.* 106

5.1 **System Versions.** *The table shows the versions of the systems that we tested with* PACE. 139
5.2 **Configurations, Workloads, and Checkers.** *The table shows the configuration, workloads and checkers for each system. We configured all systems with three nodes. The configuration settings ensure data is synchronously replicated and flushed to disk.* 141
5.3 **Vulnerabilities: File-System Requirements.** *The table shows the unique vulnerabilities categorized by file-system requirements.* . . 148
5.4 **Vulnerabilities: Consequences.** *The table shows the unique vulnerabilities categorized by user-visible consequences.* 149
5.5 **Vulnerabilities on Real File Systems.** *The table shows the number of vulnerabilities on commonly used file systems.* 151

List of Figures

2.1 **Components within a Replica.** *The figure shows distributed and local-storage components within a single replica in the system.* 18

2.2 **An Example Local-Update Protocol.** *The figure shows the local-update protocol for a simple message-insert workload in Kafka.* 20

2.3 **The Path of an Update.** *The figure shows how a write request is processed in an RSM system. As shown, the client first contacts the leader. The leader then replicates the data to the followers. When a majority of nodes have accepted the write, the leader applies the command to its state machine and returns to the client.* 22

3.1 **Sample Scenarios.** *The figure shows sample scenarios in which current approaches fail. The figure shows only the log for each server. The small boxes within a log represent individual log entries. Faulty entries are striped. Crashed and lagging nodes are shown as gray and empty boxes, respectively.* 34

3.2 **Safety Violation Example.** *The figure shows the sequence of events which exposes a safety violation with the Truncate approach.* . . 36

3.3 **Ctrl Components.** *The figure shows Ctrl's local storage and distributed recovery components.* 41

3.4 **Log Format.** *(a) shows the format of the log in a typical RSM system and the protocol used to update the log; (b) shows the same for* CLSTORE. 46
3.5 **Distributed Log Recovery.** *The figure shows how* CTRL*'s log recovery operates. All entries are appended in epoch 1 unless explicitly mentioned. For entries appended in other epochs, the epoch number is shown in the superscript. Entries shown as striped boxes are faulty. A gray box around a node denotes that it is down or extremely slow. The leader is marked with L on the left. Log indexes are shown at the top.* 53
3.6 **Leader-Initiated Identical Snapshots.** *The figure shows how leader-initiated identical snapshots is implemented in* CTRL*. The figure only shows the various states of the leader; the followers' state are not shown. (a) At first, the leader decides to take a snapshot after entry 1; hence, it inserts the* `snap` *marker (denoted by S). When the* `snap` *marker commits, and consequently when the nodes apply the marker, they take a snapshot at that moment. As shown, the snapshot operation is initiated and performed in the background. (b) While the nodes take the snapshot in the background, the leader commits entries 2 and 3 and so the in-memory state machine moves to a different state. (c) When the leader learns that a majority of nodes have taken the snapshot, it inserts the* `gc` *marker (denoted by G). (d) Finally, when the* `gc` *marker is applied, the nodes garbage collect the log entries that are part of the persisted snapshot.* 58
3.7 **CTRL Recovery Protocol Summary.** *The figure shows the summary of the protocol.* CTRL*'s recovery code is shown in thick boxes and original consensus operations are shown in dashed boxes.* 60

3.8 **Write Performance.** *(a) and (c) show the write throughput in original and* Ctrl *versions of LogCabin and ZooKeeper on an HDD. (b) and (d) show the same for SSD. The number on top of each bar shows the performance of* Ctrl *normalized to that of original.* 72

3.9 **Read Performance.** *(a) and (b) show the read throughput in original and* Ctrl *versions of LogCabin and ZooKeeper on a SSD. The number on top of each bar shows the performance of* Ctrl *normalized to that of original.* 73

4.1 **Problems in Memory-Durable Approaches.** *(a) and (b) show how a data loss or an unavailability can occur with oblivious and loss-aware memory durability, respectively. In (i), the nodes fail simultaneously; in (ii), they fail non-simultaneously, one after the other.* 80

4.2 **Summary of Protocol Behaviors and Guarantees.** *The figure shows how the disk-durable and memory-durable protocols behave under failures and the guarantees they provide.* 82

4.3 **Saucr Modes.** *The figure shows how* Saucr*'s modes work.* S_1 *is the leader. Entries in a white box are committed but are only buffered (e.g.,* e_1 *and* e_2 *in the first and second states). Entries shown grey denote that they are persisted (e.g.,* $e_1 - e_3$ *in the third state). In fast mode, a node loses its data upon a crash and is annotated with a crash symbol (e.g.,* S_5 *has lost its data in the second state).* 91

4.4 **LLE Recovery.** *The figure shows how* L' *may not be equal to* L*. For each node, we show its log and* `LLE-MAP`*. The leader (*S_1*) is marked *; crashed nodes are annotated with a crash symbol; nodes partitioned are shown in a dotted box; epochs are not shown.* 96

4.5 **Saucr Summary and Guarantees.** *The figure summarizes how* Saucr *works under failures and the guarantees it provides.* 100

4.6 **Cluster-State Sequences.** *The figure shows the possible cluster states for a five-node cluster and how cluster-state sequences are generated. One example cluster-state sequence is traced.* 104

4.7 **Micro-benchmarks.** *(a) and (b) show the update throughput on memory-durable ZK,* Saucr*, and disk-durable ZK on HDDs and SSDs, respectively. Each request is 1KB in size. The number on top of each bar shows the performance normalized to that of memory-durable ZK.* 107

4.8 **Macro-benchmarks.** *The figures show the throughput under various YCSB workloads for memory-durable ZK,* Saucr*, and disk-durable ZK for eight clients. The number on top of each bar shows the performance normalized to that of memory-durable ZK.* 110

4.9 **Performance Under Failures.** *The figure shows* Saucr*'s performance under failures; we conduct this experiment with eight clients running an update-only workload on SSDs.* 111

4.10 **Heartbeat Interval vs. Performance.** *The figure shows how varying the heartbeat interval affects performance. The left y-axis shows the average number of flushes issued by a follower per second or the average number of requests committed in slow mode by the leader per second. We measure the performance (right y-axis) by varying the heartbeat interval (x-axis). We conduct this experiment with eight clients running the YCSB-load workload on SSDs.* 112

4.11 **Correlated Failure Reaction.** *The figure shows how quickly* Saucr *reacts to correlated failures; the y-axis denotes the number of nodes that detect and flush to disk before all nodes crash when we vary the time between the individual failures (x-axis). We conduct this experiment on SSDs.* 114

5.1 **Persistent States in a Single Node.** *The figure shows file-system operations and the persistent states on a single node in a distributed system.* 120

5.2 **A Simple Distributed Protocol.** *The figure shows a simple distributed protocol. Annotations show the persistent state after performing each operation. Dash dot lines show different cuts.* 123

5.3 **PACE Workflow.** *The figure shows* Pace*'s workflow. First,* Pace *traces a workload on an initial cluster state, capturing file-system and network operations.* Pace *imposes the file system behavior through the FS APMs. Then, it uses its exploration rules to produce many distributed crash states. Finally, each distributed crash state is verified by a checker which restarts the cluster from the crash state and performs various checks (e.g., are committed data items available?). If the checker finds a violation, it reports them as vulnerabilities, pointing to the source-code lines responsible for the vulnerability.* 126

5.4 **ZooKeeper Protocol for an Update Workload.** *The figure shows the sequence of steps when the client interacts with the ZooKeeper cluster. The workload updates a value. The client prints to stdout once the update request is acknowledged.* 127

5.5 **Local File-system Update Protocol on a Single ZooKeeper Node.** *The figure shows the sequence of file-system operations on a single ZooKeeper node. Operations 1 through 6 happen on node initialization and operations 7 through 12 when the client starts interacting. Several operations that happen on initialization are not shown for clarity. (a) and (b) show two different crash scenarios.* . . . 132

5.6 **Protocols and Vulnerabilities: ZooKeeper and etcd.** *(a) and (b) show protocols and vulnerabilities in ZooKeeper and etcd, respectively. States that are not vulnerable, that were not reached in the execution, and that are vulnerable are shown by white, grey, and black boxes, respectively. The annotations show how a particular state becomes vulnerable. In Zookeeper, box (24, 24) is vulnerable because both nodes crash after the final fdatasync but before the log creation is persisted. Atomicity vulnerabilities are shown with brackets enclosing the operations that need to be persisted atomically. The arrows show the ordering dependencies in the application protocol; if not satisfied, vulnerabilities are observed. Dotted, dashed, and solid arrows represent safe file flush, directory operation, and other ordering dependencies, respectively.* 143

5.7 **Example Protocols and Vulnerabilities: Redis and Kafka.** *(a) and (b) show protocols and vulnerabilities in Redis and Kafka, respectively. Refer Figure 5.6 for legend.* 144

Abstract

We are dependent upon data in many aspects of our lives. Much of this data is stored and managed by distributed storage systems that run in data centers, powering many modern applications such as e-commerce, photo sharing, video streaming, search, social networking, messaging, collaborative editing, and even health-care and financial services.

A distributed storage system stores copies of a piece of data on many nodes for fault-tolerance: even when a few nodes fail, the system can still provide access to data. Each of these nodes depends upon a local storage stack to safely store and manage user data. The local storage stack is complex, consisting of many hardware and software components. Due to this complexity, the storage layer is a place for many potential problems to arise. This dissertation examines the reliability and performance challenges that arise the interaction points between a distributed system and the local storage stack.

In the first part of this thesis, we study how distributed storage systems react to *storage faults*: cases where the storage device may return corrupted data or errors. We focus on replicated state machine systems, an important class of distributed systems. We find that none of the existing approaches used in current systems can safely handle storage faults, leading to data loss and unavailability. Using the insights gained in our study, we design *corruption-tolerant replication* (Ctrl), a protocol-aware recovery

approach for RSM systems. CTRL exploits protocol-specific knowledge of how RSM systems operate, to ensure safety and high availability in the presence of storage faults without impacting performance.

In the second part, we study the performance and reliability properties of replication protocols used by distributed systems. We find there exists a dichotomy with respect to how and where current approaches store system state. One approach writes data to the storage stack synchronously, whereas the other buffers the data in volatile memory. The choice of whether data is written synchronously to the storage device or not greatly influences the system's robustness to crash failures and its performance. We show that existing approaches either provide robustness to crashes or performance, but not both. Thus, we introduce *situation-aware updates and crash recovery*, a dynamic protocol that, depending upon the situation, writes either synchronously or asynchronously to the storage devices, achieving both strong reliability and high performance.

In the final part of this thesis, we study the effects of file-system crash behaviors in distributed storage systems. We build *protocol-aware crash explorer* or PACE, a tool that can model and reason about file-system crash behaviors in distributed systems under a special correlated crash failure scenario. Our study reveals that the correctness of update and recovery protocols of many distributed systems hinges upon how the local file-system state is updated by each replica. We perform a detailed analysis of the vulnerabilities, showing their serious consequences and prevalence on commonly used file systems. We finally point to possible solutions to the problems discovered.

1 Introduction

We rely upon modern online services in our day-to-day lives for many purposes including social networking [229], e-commerce [161], photo sharing [187, 190, 192], video streaming [69], text messaging [195], and maintaining source repositories [191]. These modern services store massive amounts of data that include photos, documents, financial information, software, and health-care records. Most of this data is stored and managed by *distributed storage systems*. In the modern data center, these systems include MongoDB [160], Redis [189], and ZooKeeper [14].

Distributed storage systems have an important goal: to reliably store and provide efficient access to user data. To achieve this goal, a key idea used by distributed systems is that of *replication*, i.e., a piece of data is redundantly stored on many servers. Such redundancy helps mask failures: even if a few servers fail due to power-loss events, system crashes, and network failures, the data can still be accessed and updated without any problems. The resiliency of distributed systems to these kinds of failures has been thoroughly studied and is well understood [39, 202, 222].

In most modern distributed systems, the replicas work atop a local storage stack to store and manage user data. The storage stack typically consists of local file systems such as Linux ext4 [150] or Windows NTFS [211] at the top, storage devices (e.g., hard-disk drives, SSDs) at the bottom, and several software layers such as the I/O scheduler and the block layer in

between. Storage stacks are immensely complex and therefore are an important source where problems can arise [103, 186, 228].

Despite this complexity, the problems that can arise at the interaction between a distributed system and the storage subsystem remain underexamined. For instance, consider a faulty storage device on one of the replicas in a distributed system that can sometimes return corrupted data. Users and applications using the distributed system would still expect the system to return the correct data given that there are many copies of the same data on other replicas.

The interaction between the nodes of a distributed system and their local storage devices not only presents reliability challenges but also impacts performance to a great extent. For example, when committing data to the storage device, synchronously flushing the data to the storage device is considerably more expensive (about 50$\times$ slower) than buffering the data in memory and then asynchronously writing to the device. Such decisions of how the distributed system writes data to the storage subsystem significantly influence robustness to crash failures and performance.

Finally, the software layers, especially the local file systems which manage the storage devices also present challenges to a correct interaction between the distributed system and the local storage stack. Recent research has shown that file systems such as Linux ext4 and btrfs can lead to unintuitive on-disk states after a system crash or a power loss. For example, on these file systems, it is possible for the writes issued by the node to be reordered, and thus result in unexpected persistent states during crash recovery. Ideally, even if such states arise on a few nodes, causing them to lose or corrupt data, the distributed system must be able to recover the data from the redundant copies.

This dissertation is aimed at studying the reliability and performance problems that arise at the interaction points between a distributed system and the local storage stack. We perform our study in three parts. First,

we study how distributed systems react when the storage stack returns corrupted data or errors; we call these errors *storage faults*. We find that existing approaches to handling storage faults can lead to data loss or unavailability. Using the insights gained from our analysis, we devise a new approach to handle storage faults safely.

Next, we analyze the reliability and performance characteristics of current approaches to replication used in distributed systems. We find that depending upon whether these approaches synchronously write to storage or not, they either offer robustness to crashes or performance, but not both. By understanding why current approaches do not work well and how failures manifest in real deployments, we build a new replication protocol that is both performant and robust to crash failures.

Finally, we study how file-system crash behaviors affect distributed storage systems. We find that the correctness of update and recovery protocols of many distributed systems hinges upon how the local file-system state is updated by each replica. We perform a detailed analysis of the vulnerabilities and point to possible solutions.

1.1 Protocol-Aware Recovery from Storage Faults

We first examine how distributed storage systems react to storage faults: cases where portions of data persisted on a storage device could be inaccessible or corrupted on later accesses. Many distributed storage systems (especially distributed file systems) are carefully designed to handle storage faults. For example, GFS detects disk corruptions on the chunk servers and recovers the corrupted data from other replicas [94]. However, little is known about how replicated state machines (RSM) systems, an important class of distributed systems, handle storage faults.

The reliability of RSM systems is crucial: many systems entrust RSM

systems with their critical data [148]. For example, Bigtable and GFS store their metadata on Chubby [44], an RSM system; similarly, many other systems [24, 80] depend upon ZooKeeper [14], another popular RSM system. Given this, examining and understanding how RSM systems react to storage faults is valuable.

First, we conduct experimental and qualitative analyses of practical systems and methods proposed by prior research to characterize the different approaches to handling storage faults in RSM systems. The outcome of our study is the *RSM recovery taxonomy*. Our analyses show that most approaches employed by currently deployed systems do not effectively use redundancy to recover from storage faults, leading to safety violations (e.g., data loss) or unavailability. We find that a key reason why current approaches are ineffective is that they do not use any protocol-level knowledge to perform recovery.

Using the insights gained through our study, we develop a new approach that aims to effectively use redundancy to recover from storage faults. The main thesis underlying our approach is that, to correctly recover corrupted data from redundant copies in a distributed system, a recovery approach should be *protocol-aware*. A *protocol-aware recovery* (Par) approach is carefully designed based on how the distributed system performs updates to its replicated data, elects the leader, etc.

We apply the Par approach to RSM systems to improve their resiliency to storage faults. Given that many systems entrust RSM systems with their critical data, protecting these systems from storage faults such as data corruption will improve the reliability of many dependent systems. However, correctly implementing recovery is very challenging because of the strong safety and availability guarantees RSM systems provide [204]; a small misstep in recovery could violate the guarantees.

We design *corruption-tolerant replication* or Ctrl, a protocol-aware recovery approach for RSM systems that safely recovers faulty data while

providing high availability. Ctrl constitutes two components: a *local storage layer* and a *distributed recovery protocol*; while the storage layer reliably detects faults, the distributed protocol recovers faulty data from redundant copies. Both the components carefully exploit RSM-specific knowledge to ensure safety (e.g., no data loss) and high availability.

We implement Ctrl in two storage systems that are based on different consensus algorithms: LogCabin [144] and ZooKeeper [14] that are based on Raft [175] and ZAB [118], respectively. Implementation of Ctrl required only moderate developer effort and similar changes to each of the base systems, suggesting that Ctrl is easy to adopt in practical systems and applies generally to many consensus-based storage systems.

Through a series of rigorous fault-injection experiments, we show that Ctrl versions of the two systems provide safety and high availability in the presence of storage faults, while the original systems remain unsafe or unavailable in many cases. We also demonstrate that the reliability improvements of Ctrl come with little to no performance cost: Ctrl induces only about 10% overheads on HDDs and 4% on SSDs even for the worst-case workload.

1.2 Situation-Aware Updates and Crash Recovery

We next analyze the reliability and performance characteristics of different replication protocols used by distributed systems. In the modern data center, these protocols include Paxos [130], Viewstamped Replication [140], Raft [175], and ZAB [119]. The reliability and performance of these protocols are crucial: if these protocols behave incorrectly, reliability goals will not be met; if they perform poorly, excess resources and cost will be incurred.

We start by studying the existing approaches to replication in distributed systems. Our study reveals that a dichotomy exists with respect to how and where current approaches store system state. In one approach, which we call *disk durable*, critical state is replicated to persistent storage (i.e., hard drives or SSDs) within each node of the system [37, 40, 50, 112, 175]. In the contrasting *memory durable* approach, the state is replicated only to the (volatile) memory of each machine [140, 171].

We conduct performance measurements and failure-injection experiments to understand the performance and reliability characteristics of disk-durable and memory-durable protocols. Our analysis shows that neither of these approaches is ideal.

With the disk-durable approach, safety is paramount. When correctly implemented, by committing updates to disks within a majority of nodes, the disk-durable approach offers excellent durability and availability. Specifically, data will not be lost even if many or all nodes crash and recover; further, the system will remain available if a bare majority of nodes are available. Unfortunately, the cost of safety is performance. When forcing updates to hard drives, disk-durable methods incur a 50× overhead; even when using flash-based SSDs, the cost is high (roughly 2.5×).

With the memory-durable approach, in contrast, performance is generally high, but at a cost: durability. In the presence of failure scenarios where a majority of nodes crash (and then recover), existing approaches can lead to data loss or indefinite unavailability.

The distributed system developer is thus confronted with a vexing quandary: choose safety and pay a high performance cost, or choose performance and face a potential durability problem. Many systems [50, 123, 140, 171, 184] lean towards performance, employing memory-durable approaches and thus risking data loss or unavailability. Even when using a system built in a disk-durable manner, performance concerns can entice the unwary system administrator towards disaster; for instance, the

normally reliable disk-durable ZooKeeper can be configured to run in a memory-durable mode [16], leading (regrettably) to data loss [86].

Thus, to address this problem, we introduce *situation-aware updates and crash recovery* or SAUCR, a hybrid replication protocol that aims to provide the high performance of memory-durable techniques while offering strong guarantees similar to disk-durable approaches. The key idea underlying SAUCR is that the mode of replication should depend upon the situation the distributed system is in at a given time. In the common case, with many (or all) nodes up and running, SAUCR runs in memory-durable mode, thus achieving excellent throughput and low latency; when nodes crash or become partitioned, SAUCR transitions to disk-durable operation, thus ensuring safety at a lower performance level.

The effectiveness of SAUCR depends upon the simultaneity of failures. Specifically, if a window of time exists between individual node failures, the system can detect and thus react to failures as they occur. SAUCR takes advantage of this window in order to move from its fast mode to its slow-and-safe mode.

With *independent* failures, such a time gap between failures exists because the likelihood of many nodes failing together is negligible. Unfortunately, failures can often be *correlated* as well, and in that case, many nodes can fail together [88, 106, 124, 216]. Although many nodes fail together, a correlated failure does not necessarily mean that the nodes fail at the same instant: the nodes can fail either non-simultaneously or simultaneously. With non-simultaneous correlated failures, a time gap (ranging from a few milliseconds to a few seconds) exists between the individual failures; such a gap allows SAUCR to react to failures as they occur. With simultaneous failures, in contrast, such a window does not exist. However, we conjecture that such truly simultaneous failures are extremely rare; we call this the Non-Simultaneity Conjecture (NSC). While we cannot definitively be assured of the veracity of NSC, existing data [88, 96]

hints at its likely truth.

Compared to memory-durable systems, Saucr improves reliability under many failure scenarios. Under independent and non-simultaneous correlated failures, Saucr always preserves durability and availability, offering the same guarantees as a disk-durable system; in contrast, memory-durable systems can lead to data loss or unavailability. Additionally, if NSC holds, Saucr always provides the same guarantees as a disk-durable system. Finally, when NSC does not hold and if more than a majority of nodes crash in a truly simultaneous fashion, Saucr remains unavailable, but preserves safety.

We implement a prototype of Saucr in ZooKeeper. Through rigorous fault injection, we demonstrate that Saucr remains durable and available in hundreds of crash scenarios, showing its robustness. This same test framework, when applied to existing memory-durable protocols, finds numerous cases that lead to data loss or unavailability. Saucr's reliability improvements come at little or no performance cost: Saucr's overheads are within 0%-9% of memory-durable ZooKeeper across six different YCSB workloads. Compared to the disk-durable ZooKeeper, with a slight reduction in availability in rare cases, Saucr improves performance by 25× to 100× on HDDs and 2.5× on SSDs.

1.3 File-system Crash Behaviors in Distributed Systems

In the final part of this dissertation, we examine how file-system crash behaviors affect distributed systems. To safely replicate and persist data, distributed systems implement complex update protocols. For example, ZooKeeper implements an atomic broadcast protocol and several systems including LogCabin and etcd [79] implement the Raft consensus protocol to ensure agreement on data between replicas. Although the base

protocols (such as atomic broadcast [38], Raft [175], or Paxos [130]) are provably correct, implementing such a protocol without bugs is still demanding [46, 93, 104, 110, 236], especially when machines can crash at any instant [132].

When a node recovers from a crash, the common expectation is that the data stored by the node would be recoverable. Unfortunately, the local file system (which the node uses to store user data) complicates this situation. Recent research has shown that file systems vary widely with respect to how individual operations are persisted to the storage medium [181]. For example, testing has revealed that in ext4, f2fs [27], and u2fs [153], one cannot expect the following guarantee: a file always contains a prefix of the data appended to it (i.e., no unexpected data or garbage can be found in the appended portion) after recovering from a crash. The same test also shows that this property may be held by btrfs and xfs. Thus, when a node recovers from a crash, it may encounter unexpected persistent states because of the underlying file system, affecting application correctness [181].

Recent studies [41, 181] have demonstrated that these widely varying file-system behaviors influence the crash-correctness of many single-machine applications (such as SQLite [212] and LevelDB [97]), causing them lose or corrupt data. Since most practical distributed systems run atop local file systems [120, 162, 194], such file-system behaviors can affect them too.

Reasoning about file-system crash behaviors in distributed systems has a similar flavor to doing the same in a single-machine setting. However, we believe that studying the effects of file-system crash behaviors in distributed systems is a different problem for three reasons. First, distributed systems can fail in more ways than a single machine system. Since a distributed system constitutes many components, a group of components may fail together at the same or different points in the protocol.

Second, unique opportunities and problems exist in distributed crash recovery; after a failure, it is possible for one node in an inconsistent state to repair its state by contacting other nodes or to incorrectly propagate the corruption to other nodes. In contrast, single-machine applications rarely have external help. Third, crash recovery in a distributed setting has many more possible paths than single-machine crash recovery as distributed recovery depends on states of many nodes in the system.

Our goal is to examine if and how file-system crash behaviors affect distributed systems. We perform our study under a special type of correlated failure scenario in which *all* replicas of a particular data shard crash together; the individual failures can be either simultaneous or non-simultaneous. During such a failure, in most existing systems, all replicas may crash before any node can react to the failure. Such correlated failures occur due to root causes such as data-center-wide power outages [95], operator errors [239], or kernel crashes [88] and several instances of such failures have been reported [61, 67, 68, 107, 227].

When nodes recover from a correlated failure, the common expectation is that the data stored by the distributed system would be recoverable. However, file-system crash behaviors can result in unanticipated persistent states in one or more nodes when a distributed storage system recovers from a correlated crash, complicating recovery.

We say a distributed system has a *vulnerability* if a correlated crash during the execution of the system's update protocol (and subsequent recovery) exposes a user-level guarantee violation (e.g., loss of committed data). These vulnerabilities are caused by the unexpected states produced by the file system running at the replicas. To examine if file-system crash behaviors lead to vulnerabilities in distributed systems, we build PACE.

To produce states that are possible during a correlated crash, PACE considers consistent cuts in the distributed execution and generates persistent states corresponding to those cuts. PACE models local file systems at

individual replicas using an *abstract persistence model* (APM) [181] which captures the subtle crash behaviors of a particular file system. Pace uses *protocol-specific knowledge* to reduce the exploration state space by systematically choosing a subset of nodes to introduce file-system crash behaviors modeled by the APM. In the worst case, if no attributes of a distributed protocol are known, Pace can operate in a slower brute-force mode to still find vulnerabilities.

We applied Pace to eight widely used distributed storage systems spanning important domains including database caches (Redis [189]), configuration stores (ZooKeeper [14], LogCabin [144], etcd [79]), real-time databases (RethinkDB [196]), document stores (MongoDB [160]), key-value stores (iNexus [113]), and distributed message queues (Kafka [15]). Pace found a total of 26 vulnerabilities that have severe consequences such as data loss, silent corruption, and unavailability. We also find that many vulnerabilities can be exposed on commonly used file systems such as ext3, ext4, and btrfs. We reported 18 of the discovered vulnerabilities to application developers. Twelve of them have been already fixed or acknowledged by developers. While some vulnerabilities can be fixed by straightforward code changes, some are fundamentally hard to fix.

We learn two overarching lessons from our study. First, we find that file-system crash behaviors affect many distributed storage systems. For their local update protocols to work correctly, modern distributed storage systems expect certain guarantees from file systems such as ordered directory operations and atomic appends. Second, we find that in many cases, when a node loses data due to its file-system crash behaviors, the distributed recovery protocols do not use intact replicas to fix the problematic node, leading to global data loss or corruption.

Our study also demonstrates that Pace is general: it can be applied to any distributed system; Pace is systematic: it explores different systems using general rules that we develop; Pace is effective: it found 26 unique

vulnerabilities across eight widely used distributed systems.

Overall, our dissertation brings to light several reliability and performance challenges at the intersection of distributed systems and local storage stacks that were previously unexplored. Further, our thesis demonstrates that it is possible to build distributed systems that are resilient to failures that arise at the storage layer and yet deliver high performance. A key to solving these problems, as shown by our work on PAR and SAUCR, is to exploit *protocol-awareness* or *situation-awareness*, i.e., to build mechanisms that are aware of the underlying protocols used by the system or the situation in which the system is in at any particular time.

1.4 Contributions

We list the main contributions of this dissertation.

- **RSM recovery taxonomy.** We analyze existing approaches to handling storage faults in RSM systems to build the RSM recovery taxonomy. This taxonomy categorizes existing approaches into protocol-oblivious and protocol-aware classes and characterizes the safety, availability, performance, and complexity of the approaches.

- **Safety Violations in Practical Systems.** In our study on analyzing how RSM systems react to storage faults, we examine practical systems such as ZooKeeper and LogCabin using a fault-injection framework. Our tests have revealed safety violations (i.e., data loss) in these systems that were previously unknown.

- **Corruption-Tolerant Replication.** We design corruption-tolerant replication (CTRL), a new protocol-aware recovery approach for RSM

systems. Ctrl safely recovers faulty data while providing high availability. Ctrl does so with minimal performance overheads.

- **Disentangling Crashes from Corruptions.** In our work on building Ctrl, we devise a new mechanism in the local storage layer to distinguish checksum mismatches that arise due to system crashes and storage corruptions. Without such a mechanism, several existing systems conflate crashes and corruption, leading to a data loss [92]. We also present an impossibility result showing that disentanglement is not possible when the last entry in the log is corrupted. This result applies not only to Ctrl but to any log-based storage system.

- **Dichotomy in Existing Replication Protocols.** We analyze existing approaches to replication and identify that there exists a dichotomy with respect to how and where current approaches store system state (disk-durable vs. memory-durable). Through careful experimentation and analysis, we show the tradeoff between resiliency to crashes and performance in disk-durable and memory-durable approaches.

- **Situation-Aware Updates and Recovery.** We design Saucr, a new, dynamic replication protocol that provides strong durability guarantees (similar to disk-durable protocols) while delivering high performance (similar to memory-durable protocols).

- **PACE Tool.** We design and build Pace, a tool that can examine how local file-system crash behaviors affect distributed storage systems under correlated crash scenarios. Pace's source code is publicly available [6].

- **Vulnerabilities Discovered.** Using Pace, we discover 26 new vulnerabilities across eight modern distributed storage systems, including ZooKeeper, MongoDB, and Redis. Many of these vulnerabili-

ties have been acknowledged and fixed, improving the reliability of these systems. We also present a detailed study of these vulnerabilities.

1.5 Overview

We briefly describe the contents of the chapters of this dissertation.

- **Background.** In Chapter 2, we provide an overview of modern distributed storage systems. We describe the general architecture of RSM systems, how requests are processed, and how failures are handled. We provide a background on the failure models (storage faults, crash failures, and file-system crash behaviors) that are relevant to this dissertation and explain why it is critical for distributed storage systems to handle these failures.

- **Recovering from Storage Faults via Protocol-Awareness.** In Chapter 3, we first describe our analysis of current approaches to handling storage faults in RSM systems. We then present CTRL, a new protocol-aware recovery approach for RSM systems. We describe CTRL's design, implementation, and present our evaluation.

- **Situation-Aware Updates and Crash Recovery.** Chapter 4 first describes how existing approaches to replication work and shows the tradeoffs involved. Then, in the second part, we present SAUCR, a new, dynamic replication approach. We present SAUCR's design, implementation, and evaluation.

- **File-system Crash Behaviors in Distributed Systems.** In Chapter 5, we explore how file-system crash behaviors affect distributed systems. We describe how PACE explores different states that can occur in an execution and systematically introduces file-system crash

behaviors. We then present our study of vulnerabilities discovered with Pace.

- **Related Work.** In Chapter 6, we first describe prior work on studying the effects of storage faults. We then discuss prior attempts that aim to harden systems against storage faults. We discuss work related to the specific techniques used in Saucr. We finally discuss efforts related to Pace that aim to find vulnerabilities in distributed systems.

- **Conclusions and Future Work.** In Chapter 7, we summarize the dissertation and present a few high-level lessons that we learned during the course of this dissertation. We finally present directions in which the work presented in this dissertation can be extended.

2 Background

In this chapter, we provide a background on various topics relevant to this dissertation. First, in Section §2.1, we provide an overview of modern distributed storage systems and the components of a typical system. Next, in Section §2.2, we discuss how a special and an important class of distributed systems called replicated state machines works; we describe the general architecture, the path of an update, persistent structures used, and how failures are handled. Then, in Section §2.3, we describe the various failure models that are relevant to our work: storage faults, crash failures, and file-system crash behaviors.

2.1 Modern Distributed Storage Systems

Modern distributed storage systems are central to building large-scale services [52, 62, 72, 170]. Important applications such as photo stores, e-commerce, video stores, text messaging, and social networking are built upon modern distributed storage systems. By providing replication, fault tolerance, availability, and reliability, distributed storage systems ease the development of complex software services [47, 78, 159, 188].

A few examples of such modern distributed storage systems include Redis [189], MongoDB [160], ZooKeeper [14], etcd [79], LogCabin [144], Kafka [15], and RethinkDB [196]. Applications and services use these sys-

tems for different purposes; for example, as a database cache (Redis), as a key-value store (MongoDB), as a configuration store (etcd, ZooKeeper), as a message queue (Kafka), and as a real-time database (RethinkDB).

One of the main goals of these systems is to reliably store and provide efficient access to massive amounts of data. For this purpose, these systems usually *shard* the data, i.e., the data is partitioned into smaller units that can be managed independently. Each shard, in turn, is *replicated*: a data item is redundantly stored on several nodes for fault-tolerance. A node that stores a copy of data is called a *replica*. In this dissertation, we mostly focus on problems within a single shard of the distributed system.

Within a shard, most systems designate one node as the *leader* (sometimes also called the primary or master), a special node that performs a few more tasks than the other nodes. Other nodes are referred to as *followers* or backups. For example, in Redis, the leader processes updates from the clients and replicates them to the backups; the backups passively follow the leader and help distribute the load of read requests.

Components within a Replica

Figure 2.1 shows the typical components of a replica in the distributed system. As shown, a replica has two important components: distributed protocols and local storage layer. While practical systems implement many distributed protocols such as membership management [72] and reconfiguration [145], we focus on the two most pertinent to this dissertation: the *replication* protocol and the *leader-election* protocol.

Replication and Leader-election Protocols

Each replica runs a distributed replication protocol. The replication protocol determines the exact way in which data is written to the system. Typically, when a client wishes to update a piece of data, it sends a request

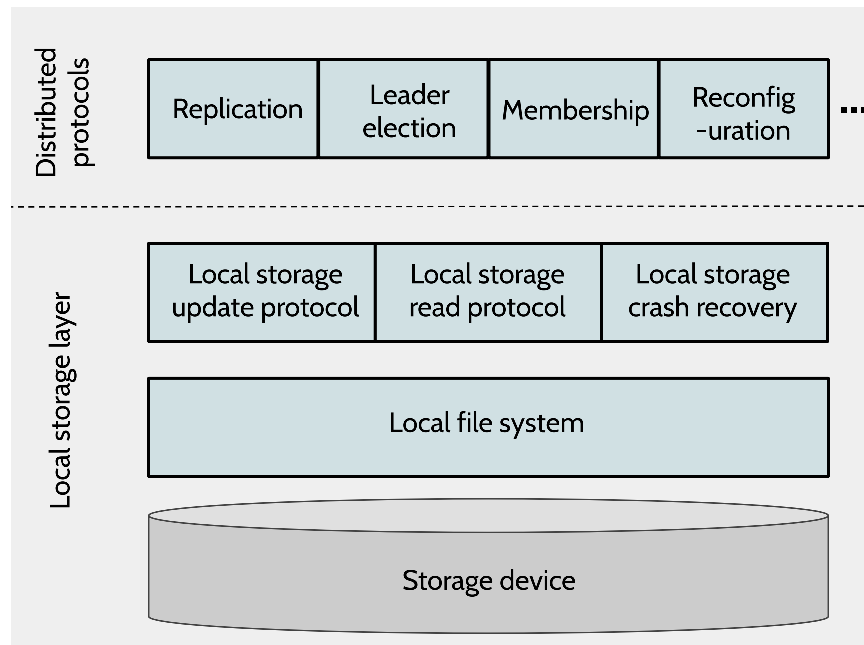


Figure 2.1: **Components within a Replica.** *The figure shows distributed and local-storage components within a single replica in the system.*

to the leader. The leader then replicates this update to the followers; once enough nodes have acknowledged the update, the leader responds to the client; at this point, the update is considered to be *committed*.

The number of replicas that need to store the data before responding to the clients may be configurable depending upon the system. For example, in MongoDB, one can configure the system to return the write just after the leader has stored the data (by setting the `writeConcern` [165] parameter appropriately). In contrast, in a few systems such as ZooKeeper and etcd, this value is fixed and cannot be changed: a write is acknowledged only after a *majority* of nodes (i.e., $\lfloor n/2 \rfloor + 1$, where n is the total number of replicas, usually a small odd number) have stored the data. In addition to configuring how many nodes must accept a write before acknowledging, most systems allow the persistence of the update to be configured too. For example, in ZooKeeper, if the *forceSync* flag is set,

then the update is persisted to disks before returning, whereas disabling this flag returns the write after the nodes have buffered the update in their volatile memory [16].

In addition to the replication protocol, each node runs a leader-election protocol. This protocol helps the nodes to constantly monitor the state of the system to detect if the current leader has failed. If the current leader fails, the protocol elects a new leader. While the details of how such an election works may vary depending on the system, a common mechanism used to detect failures is that of *heartbeats*: a small packet that a server periodically sends to notify other servers that it is alive. If a node fails (e.g., crashes or gets disconnected from the network), then the other nodes will notice the failure through missing heartbeats.

Local Storage Stack

While the distributed protocols deal with communication across different replicas and clients, the storage layer on each node is responsible for managing the data stored locally on the node. Every replica has a local storage device such as a hard-disk drive (HDD) or a flash-based solid-state drive (SSD) attached to it. The device is usually managed by a local file system such as ext4 or btrfs.

To store user data, a few systems use existing data formats such as log-structured merge trees or LSMs [172] (e.g., iNexus [113] uses LevelDB which in turn implements an LSM), while others use their custom format (e.g., ZooKeeper uses a simple custom append-only log). Most modern distributed systems store these formats on the local file system and work atop the file system to access and update data [120, 162, 194]. While a few systems manage their data directly atop the storage devices [49], in this dissertation, we mostly focus on systems that depend upon local file systems for storage. However, the solutions we develop in this dissertation apply to systems that work directly atop storage devices too.

```
mkdir(t) // create topic directory
creat(t/log) // create message log file
append(t/log) // append message to log
fsync(t/log) // persist the log
creat(rep.tmp) // create replication information temporary file
append(rep.tmp) // append replication offset info to a temporary file
fsync(rep.tmp) // persist the temp file
rename(rep.tmp, rep) // atomically rename the temp file to original
```

Figure 2.2: **An Example Local-Update Protocol.** *The figure shows the local-update protocol for a simple message-insert workload in Kafka.*

The distributed system interacts with the local file system when it needs to read or write its data. The file system takes care of safely updating its internal metadata structures through mechanisms such as journaling [28, 185]; this ensures that the file system remains consistent even in the face of system crashes or power failures. Similarly, the distributed system, which is a client to the local file system, must safely update its application-level data even in the presence of failures. For this purpose, each node implements a local *update protocol*. A correct update protocol ensures that the application-level data remains consistent and durable even in the presence of system crashes and power failures.

The update protocol is essentially a sequence of file-system-related system calls (e.g., `open`, `write`, `read`, `close`). For example, in a distributed key-value store, upon a `put` operation, each replica may issue a `write` system call to send the data to the OS buffers (specifically, the page cache) and also a `fsync` system call to force the data to the underlying storage device. Figure 2.2 shows an example update protocol implemented by Kafka. As shown, the node issues a sequence of system calls upon an update. Specifically, the update protocol creates new directories and files, writes data to the files, and carefully flushes them to the disk.

Similar to the update protocol, each node also implements a protocol to read its data from the file system. However, most of the time, the data is cached in the application's memory, requiring no interaction with the

file system during reads. Finally, each node also implements a local *recovery protocol*: the sequence of steps a node performs when it restarts after a crash to recover its data from its local file system. A typical recovery protocol first reads the files from the system's data directory; it then scans the data, performing integrity checks (e.g., sanity and checksum verification); next, it may discard any partially updated data structures and files (e.g., truncate a partially written entry in the write-ahead log); finally, it loads a portion of data into memory to serve requests. Constructing correct update and recovery protocols is often challenging, especially when the storage device or the local file system do not work in expected ways.

2.2 Replicated State Machine Systems

In Chapter 3, we focus on a special and an important class of distributed systems called the replicated state machine (RSM) systems. We thus provide an overview of RSM systems now.

RSMs provide a paradigm to make a program or state machine more reliable. The main idea is to run copies of the program on many nodes at the same time. Clients interact with the system to get their commands accepted and executed on the state machine. Given that there are many replicas, even if a few replicas fail (e.g., due to a power loss), the system as a whole can still continue to provide service to clients. If all the replicas start from the same initial state and apply the same set of inputs in the same order, then they will all produce the same outputs. A consensus algorithm such as Paxos [130], Raft [175], or ZAB [119] ensures that the replicas all process the incoming commands from the clients in the same order. Practical systems implement these protocols using a leader-based approach: a single node, the leader, establishes the order of requests and then replicates them to the followers.

RSM systems provide the following guarantees about *availability*: as

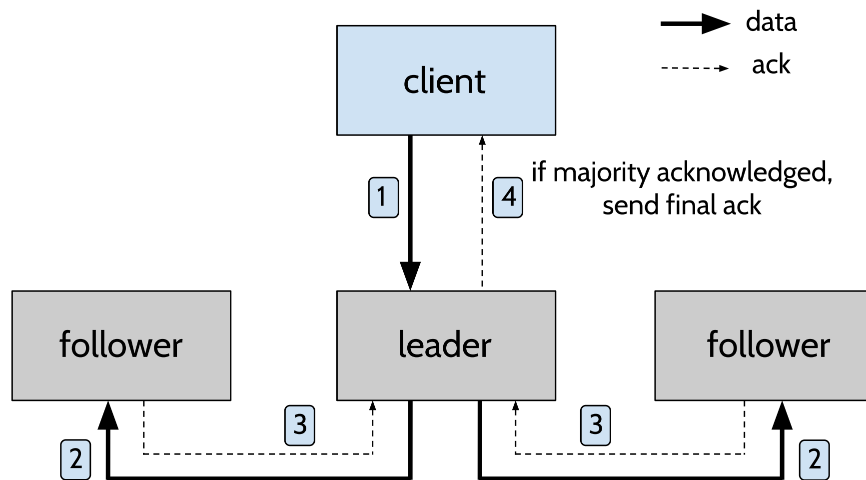


Figure 2.3: **The Path of an Update.** *The figure shows how a write request is processed in an RSM system. As shown, the client first contacts the leader. The leader then replicates the data to the followers. When a majority of nodes have accepted the write, the leader applies the command to its state machine and returns to the client.*

long as any majority of the servers are alive and can communicate with each other and with clients, then the system will remain available. For example, in a five-node cluster, the system will remain available as long as any three nodes are operational. RSM systems also provide the following *safety* property: if a command from a client has been acknowledged as committed, then the system guarantees that the command will not be lost or be overwritten. If a committed command is lost or overwritten, we say that the system has violated the safety property.

2.2.1 The Path of an Update Request

Figure 2.3 shows the path of a single write operation. In most practical RSM systems, as shown in step 1, clients interact with the leader to execute operations on the state machine. If a write request is received by a follower, it is simply redirected to the leader. Upon receiving a com-

mand, the leader stores it on its disk (for example, using a similar protocol shown in Figure 2.2) and replicates it to the followers as shown in step 2. The followers also write the data to their disks and respond back as in step 3. Once a majority of nodes inform the leader that they have written the command, the leader considers the command to be committed; thus, it applies the command to its state machine and returns the result to the client. At this point, the client expects that the command will not be lost or overwritten, regardless of any failures.

2.2.2 Persistent Structures

In Chapter 3 of this thesis, we examine the effects of data corruption in the persistent structures of an RSM system. Thus, we briefly describe the various on-disk structures that RSM systems maintain. First, the nodes append the incoming commands to an on-disk *log*. The log is usually implemented as a set of files on the file system. The update performance of the log is critical because these updates happen in the critical path of the client write request. The nodes apply the commands to their state machines in the same order as they appear in their logs.

Technically, the logs are enough to implement the state machine. But there are two problems to using only the log: first, the log can grow indefinitely and thus can exhaust disk space; second, on a restart, the node has to apply a huge number of commands stored in the log, starting from the first one, increasing recovery time. To prevent these problems, practical systems take a *snapshot* of the in-memory state machine periodically; once the snapshot is written to disk, the log entries can be garbage collected. When a node restarts after a crash, it restores the system state by reading the latest on-disk snapshot and the log. The snapshot is typically a blob stored in a file.

In addition to the log and the snapshots, the nodes also maintain a few critical metadata structures (e.g., log-start index). We call these structures

metainfo. The metainfo is usually only a few tens of bytes in size and is updated infrequently (e.g., when the current leader fails).

2.2.3 Handling Failures

As we discussed, an RSM system must remain available even when a few nodes fail, as long as a majority of servers are available. When a follower fails, and if the leader still holds a majority (including self), then availability is not affected in any way. However, when the active leader fails, the system may be momentarily unavailable until a new leader is elected. On a leader failure, the followers will start missing heartbeats from the leader. After a short time, the followers become candidates and run for an election. Most practical systems restrict which candidates can become the leader by enforcing the leader-completeness property [12, 175]; with this property, only a candidate that has stored all data items that have been acknowledged can become the leader. Once a new leader is elected, the system becomes available and can start processing requests. As we will see, this property is important to a few aspects of the solutions that we build in this dissertation.

2.3 Failure Models

One of the main contributions of this dissertation is the analysis of how modern distributed storage systems react to various realistic failures. In this section, we provide an overview of the failure models that we consider.

2.3.1 Storage Faults

The data stored by a node in the distributed system ultimately resides on a storage device such as a hard-disk drive or a flash-based SSD. Ideally,

these devices should be perfectly reliable: they should be able to retrieve the data that was stored on them without any problems.

Unfortunately, in reality, this is not the case. Storage devices exhibit a subtle and complex failure model: a few blocks of data could become inaccessible or be silently corrupted [29, 31, 99, 206]. Storage faults occur due to several reasons: media errors [32], program/read disturbance [207], and bugs in firmware [31], device drivers [218], and file systems [89, 90].

Storage faults manifest in two ways: *block errors* and *corruption*. Block errors (or latent sector errors) arise when the device internally detects a problem with a block and throws an error upon access. Studies of both hard-disk drives [32, 206] and flash-based SSDs [100, 207] show that block errors do occur in the real world. For example, a previous study [32] of one 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. Similarly, a recent study on flash reliability [207] has shown that as high as 63% and 2.5% of millions of flash devices experience at least one read and write error, respectively.

Block corruptions are more insidious than errors because blocks become corrupt in a way not detectable by the device itself. Corruption could occur due to lost and misdirected writes. Studies [31, 177] and anecdotal evidence [114, 116, 198] show the prevalence of data corruption in the real world. For example, Bairavasundaram et al., in a study of 1.53 million disk drives, showed that more than 400,000 blocks had checksum mismatches [31].

Many local file systems, on encountering a storage fault, simply propagate the fault to applications [33, 186, 214]. For example, ext4 silently returns corrupted data if the underlying device block is corrupted. In contrast, a few file systems transform an underlying fault into a different one; for example, btrfs returns an error to applications if the accessed block is corrupted on the device. In either case, storage systems built atop

local file systems should handle corrupted data and storage errors to preserve end-to-end data integrity.

Although such storage faults are rare compared to whole-machine failures, in large-scale distributed systems, even rare failures become prevalent [207, 209]. Further, these faults are more prone to occur on inexpensive disk that many distributed deployments use [70, 94]. Thus, it is critical for distributed storage systems to reliably detect and recover from storage faults.

2.3.2 Crash Failures

Most practical distributed systems intend to tolerate only fail-recover failures [98, 111, 130, 175] and not Byzantine failures [48, 131].

In the fail-recover model, nodes may crash any time (e.g., due to a power loss) and recover later (e.g., when the power is restored). When a node recovers, it loses all its volatile state and is left only with its on-disk data. In addition to crashing, sometimes, a node could be partitioned and may later be able to communicate with the other nodes; however, during such partition failures, the node does not lose its volatile state.

Node crashes happen in different ways in a data-center environment. Sometimes, a crash event could be *independent*. For example, in large deployments, single-node failure events are often independent: a crash of one node (e.g., due to a power failure) does not affect some other node. The replicas of a system are usually placed in a failure-aware manner in different availability zones, a feature common in modern data-center deployments [13, 127, 156]; thus, in many cases, the failure of one replica is independent of other replicas. With independent failures, a time gap between individual failures exists because the likelihood of many nodes failing together is negligible.

However, a more vexing failure scenario is that of *correlated crashes*: cases where many or all replicas of a system crash together and recover

at a later point [88, 106, 124, 216]. Correlated failures do occur in the real world and several instances of such failures have been reported [36, 58, 61, 67, 68, 88, 107, 125, 169, 227].

These failures typically occur due to cluster-wide power outages, planned reboots, and unplanned restarts. A recent study from Google [88] showed that node failures in Google data centers are often correlated. From data over a period of three months, the study showed that a failure burst in one instance of a distributed file system [94] can take down as many as 50 machines in a correlated fashion; this kind of failure typically can be seen during a power outage in a data center. Similarly, rolling kernel upgrades and unplanned restarts also cause failure bursts that can take down around 20 machines together.

Although many nodes fail together, a correlated failure does not necessarily mean that the nodes fail at the same instant: the nodes can fail either *non-simultaneously* or *simultaneously*. With non-simultaneous correlated failures, a time gap (ranging from a few milliseconds to a few seconds) exists between the individual failures. In contrast, with simultaneous failures, such a window between individual failures does not exist. However, we conjecture that such truly simultaneous failures are extremely rare; we explain the simultaneity of failures in more detail in Section §4.1.3.

A distributed system may not be able to progress when many or all replicas crash during a correlated failure; for example, if a majority of nodes crash in an RSM system, the system will not be able to serve client requests. However, the common expectation is that the system would become available after the cause of the failure is fixed (for example, after power has been restored) and once enough nodes have recovered. Further, applications and users will expect that the data stored by the storage system will be recoverable after the system comes alive.

2.3.3 File-system Crash Behaviors

As we discussed, most distributed storage systems depend upon local file systems such as ext4 or btrfs to store and manage their data. We also discussed how these systems implement intricate update protocols to safely update their on-disk data structures. Upon a system crash or a power failure, the system recovers its data from the file system. Unfortunately, this seemingly simple process of updating and recovering data is complicated by local file systems [181].

The complication arises because local file systems provide unclear guarantees to applications in the presence of crashes [224]. Such unclear semantics lead to wrong update and recovery protocols, causing applications to lose or corrupt data. To worsen the situation, the guarantees provided upon a crash differ from one file system to the other, and sometimes even across different mount options of the same file system. Thus, an application update and recovery protocol that work correctly on one file system may not work on another.

Recent research has studied file-system crash behaviors in detail and found that application-level consistency is dangerously dependent upon such file-system behaviors [54, 179, 181]. These studies categorize file-system crash behaviors into two classes of properties: atomicity and ordering. The atomicity class of properties says whether a particular file system must persist a particular operation in an atomic fashion in the presence of crashes. For instance, must ext2 perform a `rename` in an atomic way or can it leave the system in any intermediate state? The ordering class of properties says whether a particular file system must persist an operation A before another operation B. For instance, must ext4 order a `link` and a `write` operation?

These studies have also shown that the atomicity and ordering properties vary widely across file systems in the presence of crashes [41, 181]. For instance, the `rename` system is atomic on many file systems but not on

ext2. Similarly, while ext4 orders directory operations and file-write operations, the same does not hold true in btrfs, which can reorder directory operations and write operations in the presence of crashes.

Since most practical distributed systems run atop local file systems, it is important for them to be aware of such behaviors. These file-system nuances can result in unanticipated persistent states on a node when it recovers from a crash. In the presence of correlated crashes, the file-system behaviors can result in unexpected persistent states in many nodes when the system recovers.

2.4 Summary

In this chapter, we presented background on topics related to this dissertation. We described modern distributed storage systems and their architecture. We introduced RSM systems and showed how write requests are processed; we briefly discussed their on-disk structures and how failures are handled. We then introduced three failures models that are relevant to this dissertation: storage faults, crash failures, and file-system crash behaviors. We defined each of the failure models and provided explanations of why they occur in practice and why it is important for distributed storage systems to handle them.

3
Recovering from Storage Faults via Protocol-Awareness

In this chapter, we examine how distributed storage systems react to storage faults. We focus on replicated state machine (RSM) systems given their importance. In the first part of this chapter, we analyze different approaches to handling storage faults in RSM systems and build the *RSM recovery taxonomy*. Our analyses show that most approaches employed by currently deployed systems do not use any protocol-level knowledge to perform recovery, leading to disastrous outcomes such as data loss and unavailability.

Thus, in the second part of this chapter, to improve the resiliency of RSM systems to storage faults, we design a new protocol-aware recovery approach that we call corruption-tolerant replication or CTRL. CTRL safely recovers faulty data while ensuring high availability. We experimentally show that the CTRL versions of two systems, LogCabin and ZooKeeper, safely recover from storage faults and provide high availability, while the unmodified versions can lose data or become unavailable. We also show that the CTRL versions have little performance overhead. This chapter is based on the paper, *Protocol-Aware Recovery for Consensus-based Storage*, published in FAST 2018 [10].

We first present our analysis of how RSM systems react to storage faults (§3.1). Then, we describe the design of CTRL (§3.2). Next, we de-

scribe Ctrl's implementation in two systems (§3.3) and present our evaluation (§3.4). Finally, we summarize and conclude (§3.5).

3.1 Analysis of Existing Approaches

In this section, we first present some background on storage faults and RSM systems. Then, we present our analysis of how RSM systems detect and recover from storage faults.

3.1.1 Storage Faults in Distributed Systems

As we discussed in §2.3.1, disks and flash devices exhibit a subtle and complex failure model where a few blocks of data could become inaccessible (*block errors*) or be silently corrupted (*corruption*). Studies of both flash and hard drives [31, 32, 100, 177, 206, 207] have shown that block errors and corruptions happen in the real world.

Many local file systems that manage the storage devices do not handle storage faults: they either propagate the fault or return an error to applications. Thus, the ultimate responsibility of preserving end-to-end integrity lies with the applications. Single-machine systems rarely have external help; they solely rely on local file systems to reliably store data. Unlike single-machine systems, distributed systems inherently store data in a replicated fashion, thus providing an opportunity to recover from storage faults.

One way to tackle storage faults is to use RAID-like storage to maintain multiple copies of data on each node. However, many distributed deployments would like to use inexpensive disks [70, 94]. Given that the data in a distributed system is inherently replicated, it is wasteful to store multiple copies on each node. Hence, it is important for distributed systems to use the inherent redundancy to recover from storage faults.

3.1.2 RSM-based Storage Systems

Our goal is to examine how RSM systems react to storage faults. As we discussed in Section 2.2, in an RSM system, a set of nodes compute identical states by executing commands on a state machine (an in-memory data structure on each node) [204]. Typically, clients interact with a single node (the leader) to execute operations on the state machine. Upon receiving a command, the leader durably writes the command to an on-disk *log* and replicates it to the followers. When a majority of nodes have durably persisted the command in their logs, the leader applies the command to its state machine and returns the result to the client; at this point, the command is committed. The commands in the log have to be applied to the state machine *in-order*. Losing or overwriting committed commands violates the safety property of the state machine. The replicated log is kept consistent across nodes by a consensus protocol such as Paxos [130], ZAB [118], or Raft [175].

Because the log can grow indefinitely and exhaust disk space, periodically, a *snapshot* of the in-memory state machine is written to disk, and the log is garbage collected. When a node restarts after a crash, it restores the system state by reading the latest on-disk snapshot and the log. The node also recovers its critical metadata (e.g., log start index) from a structure called *metainfo*. Thus, each node maintains three critical persistent data structures: the *log*, the *snapshots*, and the *metainfo*.

These persistent data structures could be corrupted due to storage faults. Practical systems try to safely recover the data and remain available under such failures [40, 50]. However, as we will show, none of the current approaches correctly recover from storage faults, motivating the need for a new approach.

3.1.3 Analysis Methodology

To understand the different possible ways to handling storage faults in RSM systems, we analyze a broad range of approaches. We perform our analysis by two means: first, we analyze practical systems including ZooKeeper, LogCabin, etcd [79], and a Paxos-based system [77] using a fault-injection framework we developed; second, we analyze techniques proposed by prior research or used in proprietary systems [40, 50].

Our fault-injection framework is based on *errfs* [92], a user-level FUSE file system that systematically injects errors and corruptions into user data blocks. At a high level, our framework works as follows. First, we initialize the system under test by inserting a few data items. We ensure that the inserted items are safely persisted on the disks of at least a majority of nodes. We then configure the system under test to run atop the *errfs* file system. We then run a workload that tries to access the data stored by the RSM system (e.g., perform a read of a committed data item). Upon such an access, *errfs* injects errors and corruptions into various user data blocks (that constitute the RSM persistent data structures). We then observe how the system reacts to and recovers from the injected fault.

3.1.4 RSM Recovery Taxonomy

Through our analysis, we classify the approaches into two categories: *protocol-oblivious* and *protocol-aware*. The oblivious approaches do not use any protocol-level knowledge to perform recovery. Upon detecting a fault, these approaches take a recovery action locally on the faulty node; such actions interact with the distributed protocols in unsafe ways, leading to data loss. The protocol-aware approaches use some RSM-specific knowledge to recover; however, they do not use this knowledge correctly, leading to undesirable outcomes. Our taxonomy is *not* complete in that there may be other techniques; however, to the best of our knowledge, we have

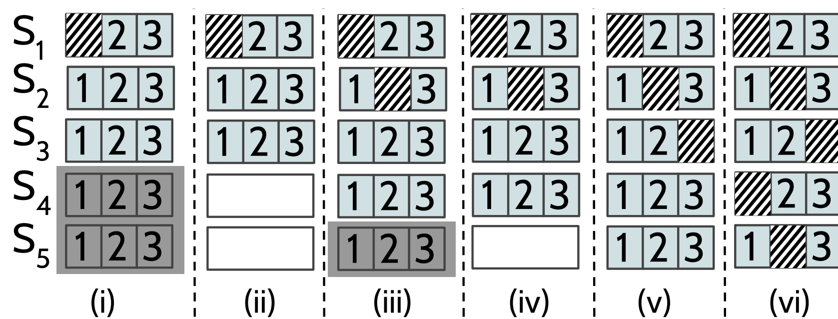


Figure 3.1: **Sample Scenarios.** *The figure shows sample scenarios in which current approaches fail. The figure shows only the log for each server. The small boxes within a log represent individual log entries. Faulty entries are striped. Crashed and lagging nodes are shown as gray and empty boxes, respectively.*

not observed other approaches apart from those in our taxonomy.

To illustrate the problems, we use Figure 3.1. In all cases, log entries[2] 1, 2, and 3 are committed; losing these items will violate safety. Table 3.1 shows how each approach behaves in Figure 3.1's scenarios. As shown in the table, all current approaches lead to safety violation (e.g., data loss), low availability, or both. A recovery mechanism that effectively uses redundancy should be safe and available in all cases. Table 3.1 also compares the approaches along other axes such as performance, maintenance overhead (intervention and extra nodes), recovery time, and complexity. Although Figure 3.1 shows only faults in the *log*, the taxonomy applies to other structures including the snapshots and the metainfo.

NoDetection. The simplest reaction to storage faults is none at all: to trust every layer in the storage stack to work reliably. For example, a few prototype Paxos-based systems [77] do not use checksums for their on-disk data; similarly, LogCabin does not protect its snapshots with checksums. *NoDetection* trivially violates safety; corrupted data can be obliviously served to clients. However, deployed systems do use checksums

[2]A log entry contains a state-machine command and data.

Class	Approach	Safety	Availability	Performance	No Intervention	No extra nodes	Fast Recovery	Low Complexity	(i)	(ii)	(iii)	(iv)	(v)	(vi)
Protocol Oblivious	*NoDetection*	×	√	√	√	√	na	√	E	E	E	E	E	E
	Crash	√	×	√	×	√	na	√	U	C	U	C	U	U
	Truncate	×	√	√	√	√	×	√	C	L	C	L	L	L
	DeleteRebuild	×	√	√	×	√	×	√	C	L	C	L	L	L
Protocol Aware	*MarkNonVoting*	×	×	√	√	√	×	√	U	C	U	C	U	U
	Reconfigure	√	×	√	×	×	×	√	U	C	U	C	U	U
	Byzantine FT	√	×	×	√	×	na	×	U	C	U	U	U	U
	CTRL	√	√	√	√	√	√	√	C	C	C	C	C	C

E- Return Corrupted, L- Data Loss, U- Unavailable, C- Correct

Table 3.1: **Recovery Taxonomy.** *The table shows how different approaches behave in Figure 3.1 scenarios. While all approaches are unsafe or unavailable,* CTRL *ensures safety and high availability.*

and other integrity strategies for most of their on-disk data.

Crash. A better strategy is to use checksums and handle I/O errors and crash the node on detecting a fault. *Crash* may seem like a good strategy because it intends to prevent any damage that the faulty node may inflict on the system. Our experiments show that the *Crash* approach is common: LogCabin, ZooKeeper, and etcd crash sometimes when their logs are faulty. Also, ZooKeeper crashes when its snapshots are corrupted.

Although *Crash* preserves safety, it suffers from severe unavailability. Given that nodes could be unavailable due to other failures, even a single storage fault results in unavailability. For instance, as shown in Fig-

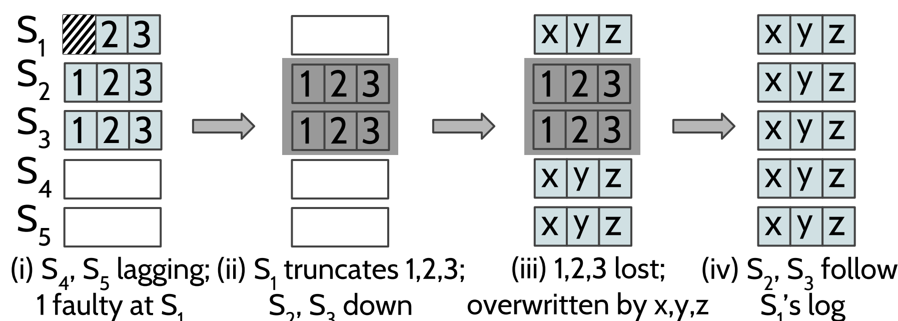


Figure 3.2: **Safety Violation Example.** *The figure shows the sequence of events which exposes a safety violation with the Truncate approach.*

ure 3.1(i), a single storage fault on S_1 causes unavailability given that the system is already operating with only a bare majority of servers. Similarly, a single fault even in different portions of data on a majority (e.g., Figure 3.1(v)) renders the system unavailable. Note that simply restarting the node does not help; storage faults, unlike other faults, could be persistent: the node will encounter the same fault and crash again until manual intervention, which is error-prone and may cause a data loss. Thus, it is desirable to recover automatically.

Truncate. A more sophisticated action is to truncate (possibly faulty) portions of data and continue operating. The intuition behind *Truncate* is that if the faulty data is discarded, the node can continue to operate (unlike *Crash*), improving availability.

However, we find that *Truncate* can cause a safety violation (data loss). Consider the scenario shown in Figure 3.2 in which entry 1 is corrupted on S_1; S_4, S_5 are lagging and do not have any entry. Assume S_2 is the leader. When S_1 reads its log, it detects the corruption; however, S_1 truncates its log, losing the corrupted entry and all subsequent entries (Figure 3.2(ii)). Meanwhile, S_2 (leader) and S_3 crash. S_1, S_4, and S_5 form a majority and elect S_1 the leader. Now the system does not have any knowledge of com-

mitted entries 1, 2, and 3, resulting in a *silent data loss*. The system also commits new entries x, y, and z in the place of 1, 2, and 3 (Figure 3.2(iii)). Finally, when S_2 and S_3 recover, they follow S_1's log (Figure 3.2(iv)), completely removing entries 1, 2, and 3.

In summary, although the faulty node detects the corruption, it truncates its log, losing the data locally. When this node forms a majority along with other nodes that are lagging, data is silently lost, violating safety. We find this safety violation in ZooKeeper and LogCabin.

Further, *Truncate* suffers from *inefficient recovery*. For instance, in Figure 3.1(i), S_1 truncates its log after a fault, losing entries 1, 2, and 3. Now to fix S_1's log, the leader needs to transfer *all* entries, increasing S_1's recovery time and wasting network bandwidth. ZooKeeper and LogCabin suffer from this slow recovery problem.

DeleteRebuild. Another commonly employed action is to manually delete all data on the faulty node and restart the node. Unfortunately, similar to *Truncate*, *DeleteRebuild* can violate safety: a node whose data is deleted could form a majority along with the lagging nodes, leading to a silent data loss. Surprisingly, administrators often use this approach, hoping that the faulty node will be "simply fixed" by fetching the data from other nodes [210, 215, 251]. *DeleteRebuild* also suffers from the slow recovery problem similar to *Truncate*.

MarkNonVoting. In this approach, used by a Paxos-based system at Google [50], a faulty node deletes all its data on a fault and marks itself as a non-voting member; the node does not participate in elections until it observes one round of consensus and rebuilds its data from other nodes. By marking a faulty node as nonvoting, safety violations such as the one in Figure 3.2 are avoided. However, *MarkNonVoting* can sometimes violate safety as noted by prior work [231]. The underlying reason for unsafety is that a

corrupted node deletes all its state including the promises[2] given to leaders. Once a faulty node has lost its promise given to a new leader, it could accept an entry from an old leader (after observing a round of consensus on an earlier entry). The new leader, however, still believes that it has the promise from the faulty node and so can overwrite the entry, previously committed by the old leader.

Further, this approach suffers from unavailability. For example, when only a majority of nodes are alive, a single fault can cause unavailability because the faulty node cannot vote; other nodes cannot now elect a leader.

Reconfigure. In this approach, a faulty node is removed, and a new node is added. However, to change the configuration, a configuration entry needs to be committed by a majority. Hence, the system remains unavailable in many cases (for example, when a majority are alive, but one node's data is corrupted). Although *Reconfigure* is not used in practical systems to tackle storage faults, it has been suggested by prior research [40, 145].

BFT. An extreme approach is to use a Byzantine-fault-tolerant algorithm which should theoretically tolerate storage faults. However, *BFT* is expensive to be used in practical storage systems; specifically, *BFT* can achieve only half the throughput of what a crash-tolerant protocol can achieve [65]. Moreover, *BFT* requires $3f + 1$ nodes to tolerate f faults [7], thus remaining unavailable in most scenarios in Figure 3.1.

[2]In Paxos, a promise for a proposal numbered p is a guarantee given by a follower (acceptor) to the leader (proposer) that it will not accept a proposal numbered less than p in the future [130].

3.1.5 Summary: The Need for Protocol-Awareness

From our analysis, we have found that none of the current approaches effectively use redundancy to recover from storage faults. Most approaches do not use any protocol-level knowledge to recover. These protocols take actions locally on the faulty node and so interact with the distributed protocol in unsafe ways, causing a global data loss or leading to unavailability. For example, the *Truncate* and *DeleteRebuild* approaches obliviously delete data on the faulty node, introducing the possibility of a user-visible data loss. Similarly, the *Crash* approach locally crashes the faulty node, leading to unavailability. Although some approaches (e.g., *MarkNonVoting*) use some RSM-specific knowledge, they do not do so correctly, causing data loss or unavailability.

Thus, to ensure safety and high availability, a recovery approach should be cognizant of how the underlying protocols used by the distributed system (e.g., leader election) operate. Such protocol-awareness can enable the recovery mechanism to effectively utilize redundancy to recover faulty data. In addition to providing safety and high availability, an ideal approach must not impact common-case performance (e.g., unlike *Byzantine FT*). Further, it must require no manual intervention or additional resources, and must recover quickly (e.g., unlike *Reconfigure*).

From the insights gained in this section, we have developed a new protocol-aware recovery approach for RSM systems that we call corruption-tolerant replication or CTRL. CTRL exploits protocol-level knowledge specific to RSM systems to ensure safely and offer high availability in the face of storage faults. CTRL's improvements come with no impact on common-case performance. Further, CTRL avoids the problems of current approaches: it requires no manual intervention and extra resources, and recovers quickly while being simple to implement. We next describe CTRL.

3.2 Corruption-Tolerant Replication

Failure recovery using redundancy is central to improved reliability of distributed systems [39, 70, 94, 108, 208, 221]. Distributed systems recover from node crashes and network failures using copies of data and functionality on several nodes [20, 163, 193]. Similarly, bad or corrupted data on one node should be recovered from redundant copies. However, unfortunately, our analyses in Section §3.1 reveal that current approaches do not effectively utilize the redundant copies to recover from storage faults, leading to undesirable outcomes such as data loss and unavailability.

In a *static* setting where all nodes always remain reachable and where clients do not actively update data, recovering corrupted data from replicas is straightforward; in such a setting, a node could repair its state by simply fetching the data from any other node.

In reality, however, a distributed system is a *dynamic* environment, constantly in a state of flux. In such settings, orchestrating recovery correctly is surprisingly hard. As a simple example, consider a quorum-based system, in which a piece of data is corrupted on one node. When the node tries to recover its data, some nodes may fail and be unreachable, some nodes may have recently recovered from a failure and so lack the required data or hold a stale version. If enough care is not exercised, the node could "fix" its data from a stale node, overwriting the new data, potentially leading to a data loss.

To correctly recover corrupted data from redundant copies in a distributed system, a recovery approach should be *protocol-aware*. A *protocol-aware recovery* (PAR) approach is carefully designed based on how the distributed system performs updates to its replicated data, elects the leader, etc. For instance, in the above example, a PAR mechanism would realize that a faulty node has to query at least R (read quorum) other nodes to safely and quickly recover its data.

As we discussed in the previous section, most current approaches in

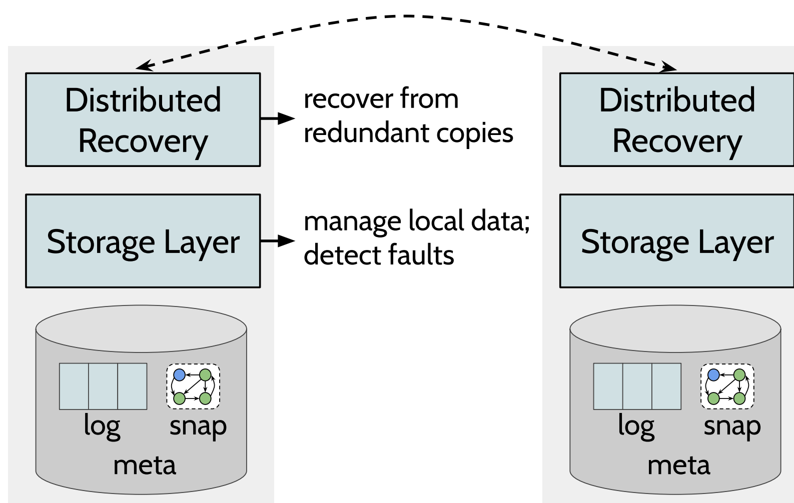


Figure 3.3: **Ctrl Components.** *The figure shows* Ctrl*'s local storage and distributed recovery components.*

RSM systems do not use protocol-level knowledge to perform recovery. Thus, we have built a new protocol-aware recovery approach for RSM systems that we call corruption-tolerant replication or Ctrl. Figure 3.3 shows an architectural overview of Ctrl. As shown, Ctrl constitutes two components: a *local storage layer* and a *distributed recovery protocol*. Ctrl divides the recovery responsibility between these two components; while the storage layer reliably detects faults, the distributed protocol recovers faulty data from redundant copies. Both the components carefully exploit RSM-specific knowledge to ensure safety (e.g., no data loss) and high availability.

Ctrl applies several novel ideas to achieve safety and high availability. First, a *crash-corruption disentanglement* technique in the storage layer distinguishes corruptions caused by crashes from disk faults; without this technique, safety violations or unavailability could result. We also establish an impossibility result regarding under what circumstances a system crash cannot be distinguished from a storage corruption. This result not

only applies to Ctrl but more generally to any log-based storage system. Next, Ctrl implements a *global-commitment determination* protocol in the distributed recovery that separates committed items from uncommitted ones. This separation is critical: while recovering faulty committed items is necessary for safety, discarding uncommitted items quickly is crucial for availability. Finally, we also design a novel *leader-initiated snapshotting* mechanism that enables bytewise identical snapshots across nodes. Such identical snapshots greatly simplify recovery.

In the remainder of this section, we first describe the protocol-level attributes that Ctrl exploits (§3.2.1). We then outline Ctrl's fault model (§3.2.2), and safety and availability guarantees (§3.2.3). We then describe the local storage layer (§3.2.4). Then, we describe Ctrl's distributed recovery in two parts: first, we show how faulty *logs* are recovered (§3.2.5) and then we explain how faulty *snapshots* are recovered (§3.2.6).

3.2.1 RSM Protocol-Level Attributes

Designing a correct recovery mechanism needs a careful understanding of the underlying protocols of the system. For example, the recovery mechanism should be cognizant of how updates are performed on the replicated data and how the leader is elected. We base Ctrl's design on the following protocol-level observations common to most RSM systems.

Leader-based. A single node acts as the leader; all data updates to the replicated data flow only through the leader.

Epochs. RSM systems partition time into logical units called *epochs*. For any given epoch, only one leader is guaranteed to exist. Every data item is associated with the epoch in which it was appended and its *index* in the log. Since the entries could only be proposed by the leader and only one leader could exist for an epoch, an $\langle epoch, index \rangle$ pair uniquely identifies a log entry.

	Fault Outcome	Possible Causes
Data	corrupted data	misdirected writes in ext-* lost writes in ext-*
	inaccessible data	latent sector errors corruptions in ZFS and btrfs
FS Metadata	missing files/directories	directory entry corrupted fsck may remove a faulty inode
	unopenable files/directories	sanity check fails on inode corruption permission bits corrupted
	files with more or fewer bytes	*i_size* field in the inode corrupted
	file system read-only	journal corrupted; fsck not run
	file system unmountable	superblock corrupted; fsck not run

Table 3.2: **Storage Fault Model.** *The table shows storage faults included in our model and possible causes that lead to a fault outcome.*

Leader Completeness. A node will not vote for a candidate if it has more up-to-date data than the candidate. Since committed data is present at least in a majority of nodes and a majority vote is required to win the election, the leader is guaranteed to have all the committed data. Although not explicitly specified in some protocols, this property is satisfied by most systems as confirmed by prior research [12, 175].

The above-listed attributes are common to most RSM system implementations. CTRL exploits these common protocol-level attributes to correctly recover from storage faults. However, CTRL cannot be readily applied to a few consensus approaches. For example, a few implementations of Paxos [166] allow updates to flow through multiple leaders at the same time. We believe CTRL can be extended to work with such RSM variants as well. We leave this extension as an avenue for future work.

3.2.2 Fault Model

Our fault model includes the standard failure assumptions made by crash-tolerant RSM systems: nodes could crash at any time and recover later, and nodes could be unreachable due to network failures [65, 142, 175]. Our model adds another realistic failure scenario where persistent data on the individual nodes could be corrupted or inaccessible. Table 3.2 shows a summary of our storage fault model. Our model includes faults in both user data and the file-system metadata blocks.

User data blocks in the files that implement the system's persistent structures could be affected by errors or corruption. A number of (possibly contiguous) data blocks could be faulty as shown by studies [35, 206]. Also, a few bits/bytes of a block could be corrupted. Depending on the local file system in use, corrupted data may be returned obliviously or transformed into errors.

File-system metadata blocks can also be affected by faults; for example, the inode of a log file could be corrupted. Our fault model considers the following outcomes that can be caused by file-system metadata faults: files/directories may go missing, files/directories may be unopenable, a file may appear with fewer or more bytes, the file system may be mounted read-only, and in the worst case, the file system may be unmountable. Some file systems such as ZFS may mask most of the above outcomes from applications [242]; however, our model includes these faulty outcomes because they could realistically occur on other file systems that provide weak protection against corruption (e.g., ext2/3/4). Through fault-injection tests, we have verified that the metadata fault outcomes shown in Table 3.2 do occur on ext4.

3.2.3 Safety and Availability Guarantees

CTRL guarantees that if there exists at least one correct copy of a *committed* data item, it will be recovered or the system will wait for that item to be fixed; committed data will never be lost. In unlikely cases where all copies of an item are faulty, the system will correctly remain unavailable. CTRL also guarantees that the system will make a decision about an *uncommitted* faulty item as early as possible, ensuring high availability.

3.2.4 CTRL Local Storage Layer

To reliably recover, the storage layer (CLSTORE) needs to satisfy three key requirements. First, CLSTORE must be able to reliably detect a storage fault. Second, CLSTORE must correctly distinguish crashes from corruptions; safety can be violated otherwise. Third, CLSTORE must identify which pieces of data are faulty; only if CLSTORE identifies which pieces have been affected, can the distributed protocol recover those pieces.

Persistent Structures Overview

As we discussed in Section §2.2, RSM systems maintain three persistent structures: the log, the snapshots, and the metainfo. CLSTORE uses RSM-specific knowledge of how these structures are used and updated, to perform its functions. For example, CLSTORE detects faults at a different granularity depending on the RSM data structure: faults in the log are detected at the granularity of individual entries, while faults in the snapshot are detected at the granularity of chunks. Similarly, CLSTORE uses the RSM-specific knowledge that a log entry is uniquely qualified by its $\langle epoch, index \rangle$ pair to identify faulty log entries.

Log. The log is a set of files containing a sequence of entries. The format of a typical RSM log is shown in Figure 3.4(a). The log is updated

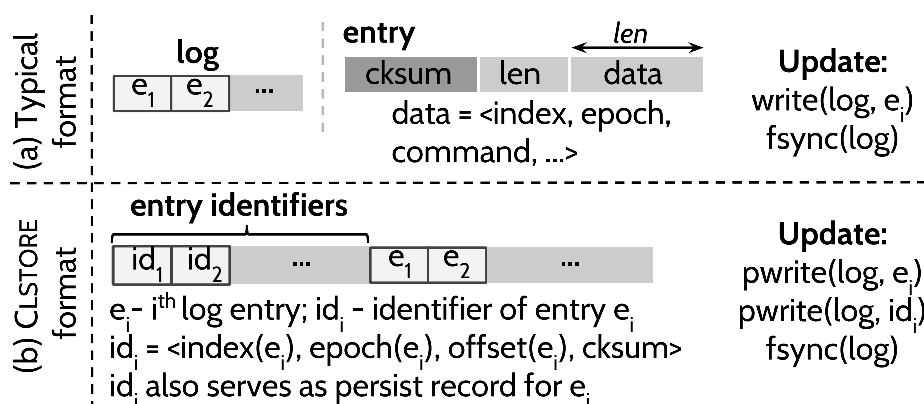


Figure 3.4: **Log Format.** *(a) shows the format of the log in a typical RSM system and the protocol used to update the log; (b) shows the same for* Clstore.

synchronously in the critical path; hence, changes made to the log format should not affect its update performance. Clstore uses a modified format as shown in Figure 3.4(b) which achieves this goal in addition to enabling detection of faulty entries. A corrupted log is recovered at the granularity of individual entries.

Snapshots. The in-memory state machine is periodically written to a snapshot. Since snapshots can be huge, Clstore splits them into chunks; a faulty snapshot is recovered at the granularity of individual chunks.

Metainfo. The metainfo is special in that faulty metainfo *cannot* be recovered from other nodes. This is because the metainfo contains information unique to a node (e.g., its current epoch, votes given to candidates); recovering metainfo obliviously from other nodes could violate safety. Clstore uses this knowledge correctly and so maintains two copies of the metainfo locally; if one copy is faulty, the other copy is used. Fortunately, the metainfo is only a few tens of bytes in size and is updated infrequently; therefore, maintaining two copies does not incur significant overheads.

Detecting Faulty Data

CLSTORE uses well-known techniques for detection: *inaccessible* data is detected by catching return codes (e.g., `EIO`) and *corrupted* data is detected by a checksum mismatch. CLSTORE assumes that if an item and its checksum agree, then the item is not faulty. In the log, each entry is protected by a checksum; similarly, each chunk in a snapshot and the entire metainfo are checksummed.

CLSTORE also handles file-system metadata faults. Missing and unopenable files/directories are detected by handling error codes upon `open`. Log and metainfo files with fewer or more bytes are detected easily because these files are preallocated and are of a fixed size; snapshot sizes are stored separately, and CLSTORE cross-checks the stored size with the file-system reported size to detect discrepancies. A read-only/unmountable file system is equivalent to a missing data directory. In most cases of file-system metadata faults, CLSTORE crashes the nodes. Crashing reliably on a metadata fault preserves safety but compromises on availability. However, we believe this is an acceptable behavior because there are far more data blocks than metadata blocks; therefore, the probability of faults is significantly less for metadata than data blocks.

Disentangling Crashes and Corruption in Log

An interesting challenge arises when detecting corruptions in the log. A checksum mismatch for a log entry could occur due to two different situations. First, the system could have *crashed* in the middle of an update; in this case, the entry would be partially written and hence cause a mismatch. Second, the entry could be safely persisted but *corrupted* at a later point. Most log-based systems conflate these two cases: they treat a mismatch as a crash [92]. On a mismatch, they discard the corrupted entry and all subsequent entries, losing the data. Discarding entries due to such

conflation introduces the possibility of a global data loss (as shown earlier in Figure 3.2).

Note that if the mismatch were really due to a crash, it is safe to discard the partially written entry. It is safe because the node would not have acknowledged to any external entity that it has written the entry. However, if an entry is *corrupted*, the entry cannot be simply discarded since it could be globally committed. Further, if a mismatch can be correctly attributed to a crash, the faulty entry can be quickly discarded locally, avoiding the distributed recovery. Hence, it is important for the local storage layer to distinguish the two cases.

To denote the completion of an operation, many systems write a commit record [37, 55]. Similarly, CLSTORE writes a persist record, p_i, after writing an entry e_i. For now, assume that e_i is ordered before p_i, i.e., the sequence of steps to append an entry e_i is $write(e_i), fsync(), write(p_i), fsync()$. On a checksum mismatch for e_i, if p_i is not present, we can conclude that the system crashed during the update. Conversely, if p_i is present, we can conclude that the mismatch was caused due to a corruption and *not* due to a crash. p_i is checksummed and is very small; it can be atomically written and thus cannot be "corrupted" due to a crash. If p_i is corrupted in addition to e_i, we can conclude that it is a corruption and not a crash.

The above logic works when e_i is ordered before p_i. However, such ordering requires an (additional) expensive *fsync* in the critical path, affecting log-update performance. For this reason, CLSTORE does not order e_i before p_i; thus, the append protocol is $\mathbf{t_1}$:$write(e_i)$, $\mathbf{t_2}$:$write(p_i)$, $\mathbf{t_3}$:$fsync()$.[2] Given this update sequence, assume a checksum mismatch occurs for e_i. If p_i is *not present*, CLSTORE can conclude that it is a crash (before $\mathbf{t_2}$) and discard e_i. Contrarily, if p_i is *present*, there are two possibilities: either e_i could be affected by a corruption after $\mathbf{t_3}$ or a crash could have occurred between $\mathbf{t_2}$ and $\mathbf{t_3}$ in which p_i hit the disk while e_i was only

[2]The final *fsync* is required for durability.

partially written. The second case is possible because file systems can reorder writes between two *fsync* operations and e_i could span multiple sectors [12, 56, 180, 181]. CLSTORE can still conclude that it is a corruption if e_{i+1} or p_{i+1} is present. However, if e_i is the *last entry*, then we cannot determine whether it was a crash or a corruption. A proof of this claim can be found in appendix A.

The inability to disentangle the last entry when its persist record is present is not specific to CLSTORE, but rather a fundamental limitation in log-based systems. For instance, in ext4's journal_async_commit mode (where a transaction is not ordered before its commit record), a corrupted last transaction is assumed to be caused due to a crash, possibly losing data [117, 226]. Even if crashes and corruptions can be disentangled, there is little a single-machine system can do to recover the corrupted data. However, in a distributed system, redundant copies can be used to recover. Thus, when the last entry cannot be disentangled, CLSTORE safely marks the entry as *corrupted* and leaves it to the distributed recovery to fix or discard the entry based on the global commitment.

The entanglement problem does not arise for snapshots or metainfo. These files are first written to a temporary file and then atomically renamed. If a crash happens before the rename, the partially written temporary file is discarded. Thus, the system will never see a corrupted snapshot or metainfo due to a crash; if these structures are corrupted, it is because of a storage corruption.

Identifying Faulty Data

Once a faulty item is detected, it has to be *identified*; only if CLSTORE can identify a faulty item, the distributed layer can recover the item. For this purpose, CLSTORE redundantly stores an *identifier* of an item apart from the item itself; duplicating only the identifier instead of the whole item obviates the (2×) storage and performance overhead. However, storing

the identifier near the item is less useful; a misdirected write can corrupt both the item and its identifier [31, 32]. Hence, identifiers are physically separated from the items they identify.

The ⟨*epoch*, *index*⟩ pair serves as the identifier for a log entry and is stored separately at the head of the log, as shown in Figure 3.4(b). The offset of an entry is also stored as part of the identifier to enable traversal of subsequent entries on a fault. The identifier of a log entry also conveniently serves as its persist record. Similarly, for a snapshot chunk, the ⟨*snap-index*, *chunk#*⟩ pair serves as the identifier; the *snap-index* and the snapshot size are stored in a separate file than the snapshot file. The identifiers have a nominal storage overhead (32 bytes for log entries and 12 bytes for snapshots), can be atomically written to disk, and are also protected by a checksum.

It is highly unlikely an item and its identifier will both be faulty since they are physically separated [31, 32, 35, 206]. In such unlikely and unfortunate cases, CLSTORE crashes the node to preserve safety. Table 3.3 (second column) summarizes CLSTORE's key techniques.

3.2.5 CTRL Distributed Log Recovery

The local storage layer detects faulty data items and passes on their identifiers to the distributed recovery layer. We now describe how the distributed layer recovers the identified faulty items from redundant copies using RSM-specific knowledge. We first describe how *log entries* are recovered and subsequently describe *snapshot* recovery. As we discussed, metainfo files are recovered locally and so we do not discuss them any further. We use Figure 3.5 to illustrate how log recovery works.

Naive Approach: Leader Restriction. RSM systems do not allow a node with an incomplete log to become the leader. A naive approach to recovering from storage faults could be to impose an additional constraint on the

election: *a node cannot be elected the leader if its log contains a faulty entry*. The intuition behind the naive approach is as follows: since the leader is guaranteed to have all committed data and our new restriction ensures that the leader is not faulty, faulty log entries on other nodes could be fixed using the corresponding entries on the leader. Cases (a)(i) and (a)(ii) in Figure 3.5 show scenarios where the naive approach could elect a leader. In (a)(i), only S_1 can become the leader because other nodes are either lagging or have at least one faulty entry. Assume S_1 is the leader also in case (a)(ii).

Fixing Followers' Logs. When the leader has no faulty entries, fixing the followers is straightforward. For example, in case (a)(i), the followers inform S_1 of their faulty entries; S_1 then supplies the correct entries. However, sometimes the leader might not have any knowledge of an entry that a follower is querying for. For instance, in case (a)(ii), S_5 has a faulty entry at index 3 but with a *different epoch*. This situation is possible because S_5 could have been the leader for epoch 2 and crashed immediately after appending an entry. As discussed earlier, an entry is uniquely identified by its ⟨*epoch*, *index*⟩; thus, when querying for faulty entries, a node needs to specify the epoch of the entry in addition to its index. Thus, S_5 informs the leader that its entry ⟨*epoch*:2, *index*:3⟩ is faulty. However, S_1 does not have such an entry in its log. If the leader does not have an entry that the follower has, then the entry *must be an uncommitted entry* because the leader is guaranteed to have all committed data; thus, the leader instructs S_5 to truncate the faulty entry and also replicates the correct entry.

Although the naive approach guarantees safety, it has availability problems. The system will be unavailable in cases such as the ones shown in (b): a leader cannot be elected because the logs of the alive nodes are either faulty or lagging. Note that even a single storage fault can cause an unavailability as shown in (b)(i). It is possible for a carefully designed re-

covery protocol to provide better availability in these cases. Specifically, since at least one intact copy of all committed entries exists, it is possible to collectively reconstruct the log.

Removing the Restriction Safely

To recover from scenarios such as those in Figure 3.5(b), we remove the additional constraint on the election. Specifically, any node that has a more up-to-date log can now be elected the leader even if it has faulty entries. This relaxation improves availability; however, two key questions arise: first, when can the faulty leader proceed to accept new commands? second, and more importantly, is it safe to elect a faulty node as the leader?

To accept a new command, the leader has to append the command to its log, replicate it, and apply it to the state machine. However, before applying the new command, *all* previous commands must be applied. Specifically, faulty commands cannot be skipped and later applied when they are fixed; such out-of-order application would violate safety. Hence, it is required for the leader to fix its faulty entries before it can accept new commands. Thus, for improved availability, the leader needs to fix its faulty entries as early as possible.

The crucial part of the recovery to ensure safety is to fix the leader's log using the redundant copies on the followers. In simple cases such as (b)(i) and (b)(ii), the leader S_1 could fix its faulty entry $\langle epoch{:}1, index{:}1\rangle$ using the correct entries from the followers and proceed to normal operation. However, in several scenarios, the leader cannot immediately recover its faulty entries; for example, none of the reachable followers might have any knowledge of the entry to be recovered or the entry to be recovered could also be faulty on the followers.

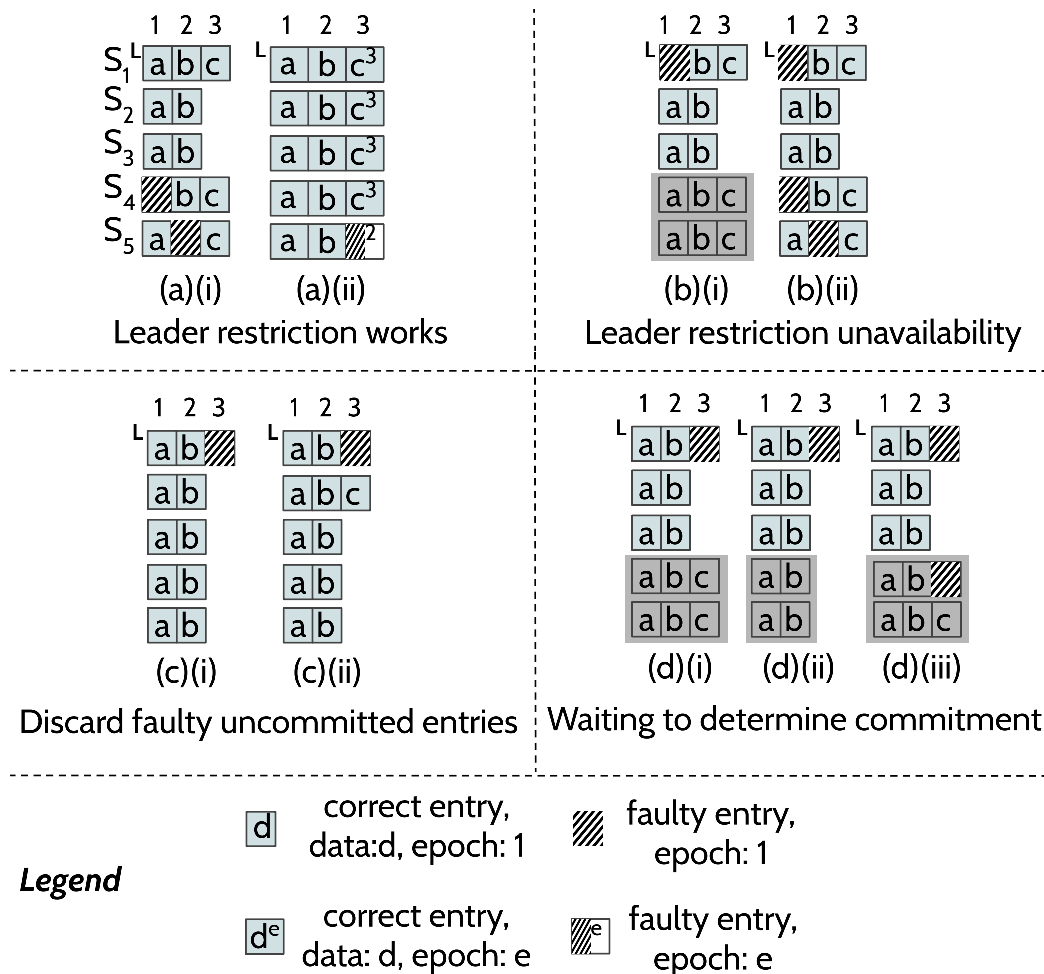


Figure 3.5: **Distributed Log Recovery.** *The figure shows how* CTRL*'s log recovery operates. All entries are appended in epoch 1 unless explicitly mentioned. For entries appended in other epochs, the epoch number is shown in the superscript. Entries shown as striped boxes are faulty. A gray box around a node denotes that it is down or extremely slow. The leader is marked with L on the left. Log indexes are shown at the top.*

Determining Commitment

The main insight to fix the leader's faulty log safely and quickly is to distinguish *uncommitted* entries from *possibly committed* ones; while recover-

ing the committed entries is necessary for safety, uncommitted entries can be safely discarded. Further, discarding uncommitted faulty entries immediately is crucial for availability. For instance, in case (c)(i), the faulty entry on S_1 cannot be fixed since there are no copies of it; waiting to fix that entry results in indefinite unavailability. Sometimes, an entry could be partially replicated but remain uncommitted; for example, in case (c)(ii), the faulty entry on S_1 is partially replicated but is not committed. Although there is a possibility of recovering this entry from the other node (S_2), this is *not* necessary for safety; it is completely safe for the leader to discard this uncommitted entry.

To determine the commitment of a faulty entry, the leader queries the followers. If a majority of the followers respond that they do *not* have the entry (negative acknowledgment), then the leader concludes that the entry is uncommitted. In this case, the leader safely discards that entry and all subsequent entries; it is safe to discard the subsequent entries because entries are committed only in order (i.e., if an entry at index i is an uncommitted entry, then all entries after i must also be uncommitted entries). Conversely, if the entry were committed, at least one node in this majority would have that entry and inform the leader of it; in this case, the leader can fix its faulty entry using that response.

Waiting to Determine Commitment. Sometimes, it may be impossible for the leader to quickly determine commitment. For instance, consider the cases in Figure 3.5(d) in which S_4 and S_5 are down or slow. S_1 queries the followers to recover its entry $\langle$*epoch*:1, *index*:3$\rangle$. S_2 and S_3 respond that they do not have such an entry (negative acknowledgment). S_4 and S_5 do not respond because they are down or slow. The leader, in this case, has to wait for either S_4 or S_5 to respond; discarding the entry without waiting for S_4 or S_5 could violate safety. However, once S_4 or S_5 responds, the leader will make a decision immediately. In (d)(i), S_4 or S_5 would

respond with the correct entry, fixing the leader. In (d)(ii), S_4 or S_5 would respond that it does not have the entry, accumulating three (a majority out of five) negative acknowledgments; hence, the leader can conclude that the entry is uncommitted, discard it, and continue to normal operation. In (d)(iii), S_4 would respond that it has the entry but is faulty in its log too. In this case, the leader has to wait for the response from S_5 to determine commitment. In the unfortunate and unlikely case where all copies of an entry are faulty, the system will remain unavailable.

The Complete Log Recovery Protocol

We now assemble the pieces of the log recovery protocol. First, fixing faulty followers is straightforward; the committed faulty entries on the followers can be eventually fixed by the leader because the leader is guaranteed to have all committed data. Faulty entries on followers that the leader does not know about are uncommitted; hence, the leader instructs the followers to discard such entries.

The main challenge is thus fixing the leader's log. The leader queries the followers to recover its entry ⟨*epoch*:e, *index*:i⟩. Three types of responses are possible:

Response 1: **have** – a follower could respond that it has the entry ⟨*epoch*:e, *index*:i⟩ and is not faulty in its log.

Response 2: **dontHave** – a follower could respond that it does not have the entry ⟨*epoch*:e, *index*:i⟩ in its log.

Response 3: **haveFaulty** – a follower could respond that it has ⟨*epoch*:e, *index*:i⟩ but is faulty in its log too.

Once the leader collects these responses, it takes the following possible actions:

Case 1: if it gets a *have* response from at least one follower, it fixes the entry in its log.

Case 2: if it gets a *dontHave* response from a majority of followers, it con-

firms that the entry is uncommitted, discards that entry and all subsequent entries.
Case 3: if it gets a *haveFaulty* response from a follower, it waits for either Case 1 or Case 2 to happen.

Case 1 and Case 2 can happen in any order; both orderings are safe. Specifically, if the leader decides to discard the faulty entry (after collecting a majority *dontHave* responses), it is safe since the entry was uncommitted anyways. Conversely, there is no harm in accepting a correct entry (at least one *have* response) and replicating it. The first to happen out of these two cases will take precedence over the other.

The leader proceeds to normal operation only after its faulty data is discarded or recovered. However, CTRL discards uncommitted data as early as possible and minimizes the recovery latency by recovering faulty data at a fine granularity (as we show in §4.4.2), ensuring that the leader proceeds to normal operation quickly.

The leader could crash or be partitioned while recovering its log. On a leader failure, the followers will elect a new leader and make progress. The partial repair done by the failed leader is harmless: it could have either fixed committed faulty entries or discarded uncommitted ones, both of which are safe.

3.2.6 CTRL Distributed Snapshot Recovery

Because the logs can grow indefinitely, periodically, the in-memory state machine is written to disk and the logs are garbage collected. Current systems including ZooKeeper and LogCabin do not handle faulty snapshots correctly: they either crash or load corrupted snapshots obliviously. CTRL aims to recover faulty snapshots from redundant copies. Snapshot recovery is different from log recovery in that all data in a snapshot is committed and already applied to the state machine; hence, faulty snapshots cannot be discarded in any case (unlike uncommitted log entries

which can be discarded safely).

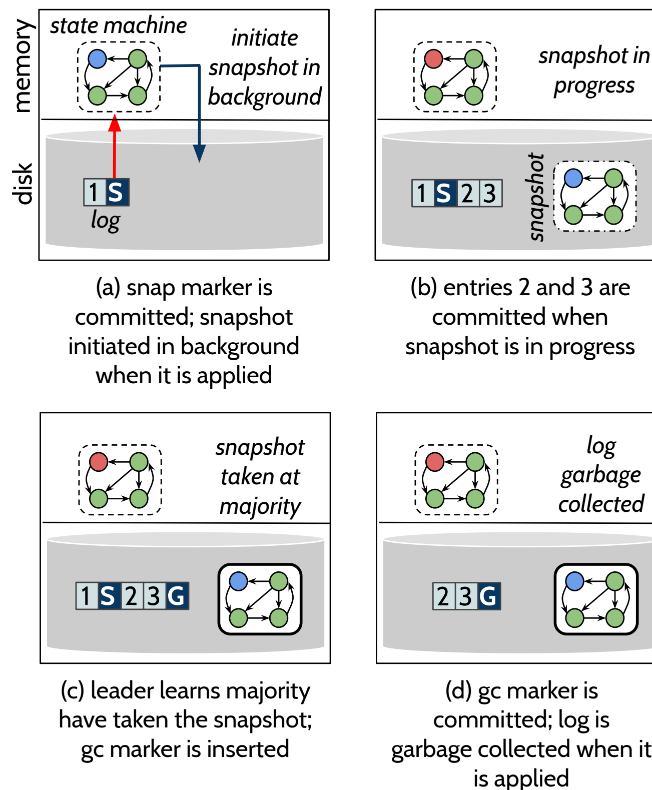
Leader-Initiated Identical Snapshots

Current systems [144] have two properties with respect to snapshots. First, they allow new commands to be applied to the state machine while a snapshot is in progress. Second, they take index-consistent snapshots: a snapshot S_i represents the state machine in which log entries exactly up to i have been applied. One of the mechanisms used in current systems to realize the above two properties is to take snapshots in a `fork`-ed child process; while the child can write an index-consistent image to the disk, the parent can keep applying new commands to its copy of the state machine. CTRL should enable snapshot recovery while preserving the above two properties.

In current systems, every node runs the snapshot procedure independently, taking snapshots at different log indexes. Because the snapshots are taken at different indexes, snapshot recovery can be complex: a faulty snapshot on one node cannot be simply fetched from other nodes. Further, snapshots cannot be recovered at the granularity of chunks because they will be byte-wise non-identical; entire snapshots have to be transferred across nodes, slowing down recovery.

This complexity can be significantly alleviated if the nodes take the snapshot at the same index; identical snapshots also enable chunk-based recovery.

However, coordinating a snapshot operation across nodes can, in general, affect the common-case performance. For example, one naive way to realize identical snapshots is for the leader to produce the snapshot, insert it into the log as yet another entry, and replicate it. However, such an approach will affect update performance since the snapshot could be huge and all client commands must wait while the snapshot commits [173]. Moreover, transferring the snapshot to the followers wastes network band-



(a) snap marker is committed; snapshot initiated in background when it is applied

(b) entries 2 and 3 are committed when snapshot is in progress

(c) leader learns majority have taken the snapshot; gc marker is inserted

(d) gc marker is committed; log is garbage collected when it is applied

Figure 3.6: **Leader-Initiated Identical Snapshots.** *The figure shows how leader-initiated identical snapshots is implemented in* CTRL. *The figure only shows the various states of the leader; the followers' state are not shown. (a) At first, the leader decides to take a snapshot after entry 1; hence, it inserts the* `snap` *marker (denoted by S). When the* `snap` *marker commits, and consequently when the nodes apply the marker, they take a snapshot at that moment. As shown, the snapshot operation is initiated and performed in the background. (b) While the nodes take the snapshot in the background, the leader commits entries 2 and 3 and so the in-memory state machine moves to a different state. (c) When the leader learns that a majority of nodes have taken the snapshot, it inserts the* `gc` *marker (denoted by G). (d) Finally, when the* `gc` *marker is applied, the nodes garbage collect the log entries that are part of the persisted snapshot.*

width.

CTRL takes a different approach to identical snapshots that preserves

common-case performance. The leader initiates the snapshot procedure by first deciding the index at which a snapshot will be taken and informing the followers of the index. Once a majority agree on the index, all nodes independently take a snapshot at the index. When the leader learns that a majority (including itself) have taken a snapshot at an index `i`, it garbage collects its log up to `i` and instructs the followers to do the same.

CTRL implements the above procedure using the log. When the leader decides to take a snapshot, it inserts a special marker called `snap` into the log. When the `snap` marker commits, and thus when a node applies the marker to the state machine, it takes a snapshot (i.e., the snapshot corresponds to the state where commands exactly up to the marker have been applied). Within each node, we reuse the same mechanism used by the original system (e.g., a `fork`-ed child) to allow new commands to be applied while a snapshot is in progress. Notice that the snapshot operation happens independently on all nodes but the operation will produce identical snapshots because the marker will be seen at the same log index by all nodes when it is committed. When the leader learns that a majority of nodes (including itself) have taken a snapshot at an index `i`, it appends another marker called `gc` for `i`; when the `gc` marker is committed and applied, the nodes garbage collect their log entries up to `i`. Figure 3.6 illustrates how leader-initiated identical snapshots works.

Recovering Snapshot Chunks

With the identical-snapshot mechanism, snapshot recovery becomes easier. Once a faulty snapshot is detected, the local storage layer provides the distributed protocol the snapshot index and the chunk that is faulty. The distributed protocol recovers the faulty chunk from other nodes. First, the leader recovers its faulty chunks from the followers and then fixes the faulty snapshots on followers. Three cases arise during snapshot recovery.

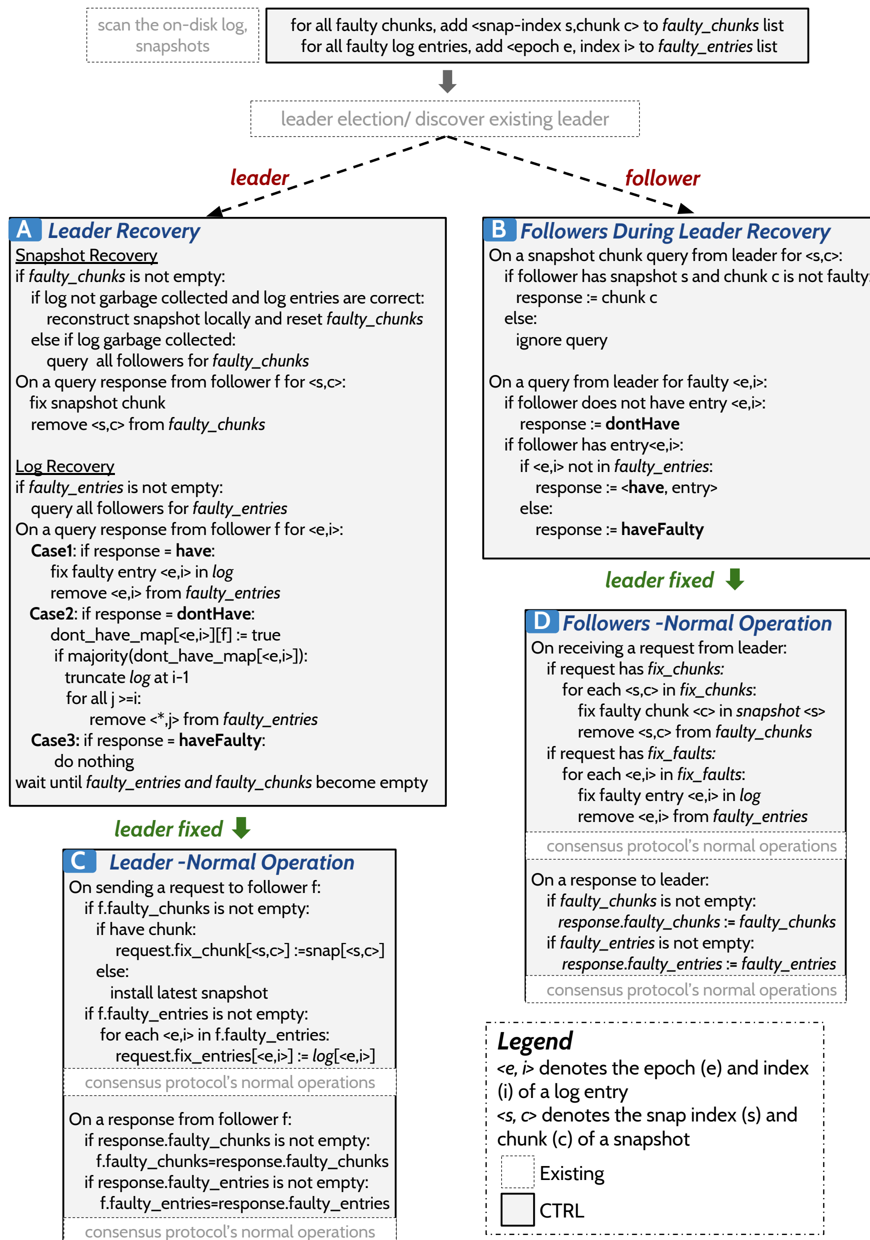


Figure 3.7: **CTRL Recovery Protocol Summary.** *The figure shows the summary of the protocol. CTRL's recovery code is shown in thick boxes and original consensus operations are shown in dashed boxes.*

	Local Storage	Distributed Recovery
Log	granularity: entry identifier:$\langle$*epoch*, *index*$\rangle$ crash-corruption disentanglement	global-commitment determination to fix leader, leader fixes followers
Snapshot	granularity: chunk identifier:$\langle$*snap-index*, *chunk#*$\rangle$ no entanglement	leader-initiated identical snapshots, chunk-based recovery
Metainfo	granularity: file identifier: n/a no entanglement	none (only internal redundancy)

Table 3.3: **Techniques Summary.** *The table shows a summary of techniques employed by* CTRL*'s storage layer and distributed recovery.*

First, the log entries for a faulty snapshot may not be garbage collected yet; in this case, the snapshot is recovered locally from the log (after fixing the log if needed).

Second, if the log is garbage collected, then a faulty snapshot has to be recovered from other nodes. However, if the log entries for a snapshot are garbage collected, then at least a majority of the nodes must have taken the same snapshot. This is true because the gc marker is inserted only after a majority of nodes have taken the snapshot. Thus, faulty garbage-collected snapshots are recovered from those redundant copies.

Third, sometimes, the leader may not know a snapshot that a follower is querying for (for example, if a follower took a snapshot and went offline for a long time and the leader replaced that snapshot with an advanced one); in this case, the leader supplies the full advanced snapshot.

3.2.7 CTRL Summary

CTRL's storage layer detects faulty data using checksums and handling er-

rors. It also disentangles crashes and corruptions in the log. Finally, it identifies which portions of the data are faulty and passes on the identifiers to the distributed recovery layer.

The distributed protocol recovers the faulty data from the redundant copies. Figure 3.7 summarizes the distributed recovery protocol. Ctrl decouples the recovery of followers from that of the leader. In all cases, fixing the followers is straightforward: the leader supplies the correct data because the leader is guaranteed to have all the committed data. Ctrl couples the fixing of followers with common-case operations such as replication of entries. Actions taken by the leader and the followers to fix the followers' data are shown in boxes C and D of Figure 3.7. The leader can fix its faulty snapshots from its local log if the log is not garbage collected yet. If the log is garbage collected, the leader recovers the snapshot from the followers (a majority of nodes are guaranteed to have the snapshot). The leader fixes its log by determining commitment of the faulty entries. Actions taken by the leader and the followers during leader recovery are shown in boxes A and B of Figure 3.7.

Ctrl's storage and distributed recovery layers exploit RSM-specific knowledge to perform their functions. Table 3.3 shows a summary of techniques employed in both the layers.

3.3 Implementation

We implement Ctrl in two different RSM systems, LogCabin (v1.0) and ZooKeeper (v3.4.8); while LogCabin is based on Raft, ZooKeeper is based on ZAB. Implementing Ctrl's storage layer and distributed recovery took only a moderate developer effort; Ctrl adds about 1500 lines of code to each of the base systems.

3.3.1 Local Storage Layer

We implemented CLSTORE by modifying the storage engines of LogCabin and ZooKeeper. In both systems, the log is a set of files, each of a fixed size and preallocated with zeros. The header of each file is reserved for the log-entry identifiers. The size of the reserved header is proportional to the file size. CLSTORE ensures that a log entry and its identifier are at least a few megabytes physically apart. Both systems batch many log entries to improve update performance. With batching, CLSTORE performs crash-corruption disentanglement as follows: the first faulty entry without an identifier and its subsequent entries are discarded; faulty entries preceding that point are marked as corrupted and passed on to the distributed layer.

In both systems, the state machine is a data tree. We modified both the systems to take index-consistent identical snapshots: when a `snap` marker is applied, the state machine (i.e., the tree) is serialized to the disk. The *snap-index* and snapshot size are stored separately. CLSTORE uses a chunk size of 4K, enabling fine-grained recovery.

In LogCabin, the metainfo contains the `currentTerm` and `votedFor` structures. Similarly, in ZooKeeper, structures such as `acceptedEpoch` and `currentEpoch` constitute the metainfo. CLSTORE stores redundant copies of metainfo and protects them using checksums.

Log entries, snapshot chunks, and metainfo are protected by a CRC32 checksum. CLSTORE detects inaccessible data items by catching errors (`EIO`); it then populates the item's in-memory buffer with zeros, causing a checksum mismatch. Thus, CLSTORE deals with both corruptions and errors as checksum mismatches. Lost log writes result in checksum mismatches because the log is preallocated with zeros. Misdirected writes can overwrite previously written log entries. Such misdirected writes typically occur at the block or sector granularity, causing a checksum mismatch for the log entries in most cases. In rare cases, the entries could be block

aligned, and a misdirected write may not cause a checksum mismatch. However, the storage layer catches such cases through a sanity check that verifies that the index of the log entries are in order and are monotonically increasing.

3.3.2 Distributed Recovery

LogCabin. In Raft, *terms* are equivalent to epochs. Thus, a log entry is uniquely identified by its $\langle term, index \rangle$ pair. To fix the followers, we modified the AppendEntries RPC used by the leader to replicate entries [175]. The followers inform the leader of their faulty log entries and snapshot chunks in the responses of this RPC; the leader sends the correct entries and chunks in a subsequent RPC. A follower starts applying commands to its state machine once its faulty data is fixed. To fix the leader, we added a new RPC which the leader issues to the followers. The leader does not proceed to normal operation until its faulty data is fixed. After a configurable recovery timeout, the leader steps down if it is unable to recover its faulty data (for example, due to a partition), allowing other nodes to become the leader. Several entries and chunks are batched in a single request/response, avoiding multiple round trips.

ZooKeeper. In ZAB, the epoch and index are packed into the *zxid* which uniquely identifies a log entry [17]. Followers discover and connect to the leader in Phase 1. We modified Phase 1 to send information about the followers' faulty data. The followers are synchronized with the leader in Phase 2. We modified Phase 2 so that the leader sends the correct data to the followers. The leader waits to hear from a majority during Phase 1 after which it sends a `newEpoch` message; we modified this message to send information about the leader's faulty data. The leader does not proceed to Phase 2 until its data is fixed.

3.4 Evaluation

We evaluate the correctness and performance of CTRL versions of LogCabin and ZooKeeper. We conducted our performance experiments on a three-node cluster on a 1-Gbps network; each node is a 40-core Intel Xeon CPU E5-2660 machine with 128 GB memory running Linux 3.13, with a 480-GB SSD (Intel SSDSC2BB480G7K) and a 1-TB HDD (Seagate ST1200MM0088) managed by ext4. We configure LogCabin and ZooKeeper to store their persistent structures on this ext4 partition.

3.4.1 Correctness

To test CTRL's safety and availability guarantees, we built a fault-injection framework that can inject storage faults (targeted corruptions and random block corruptions and errors). The framework can also inject crashes. By injecting crashes at different points in time, the framework simulates lagging nodes. After injecting faults, we issue reads from clients to determine whether the target system remains available and preserves safety.

We first exercise different log-recovery scenarios. Then, we test snapshot recovery, and finally file-system metadata fault recovery.

Log Recovery

We perform three different experiments to test log-recovery: targeted log-entry corruptions, random block corruptions and errors, and faults with crashed and lagging nodes.

Targeted Corruptions. We initialize the cluster by inserting four log entries and ensuring that the entries are replicated to all three nodes in the cluster. We exercise all combinations of entry corruptions across the three nodes ($(2^4)^3 = 4096$ combinations). Out of the 4096 cases, a correct recovery is possible in 2401 cases (at least one non-faulty copy of each entry

System	Recovery Scenario	Total Test Cases	Original Approach	Original Outcomes: Unavailable	Original Outcomes: Unsafe	Original Outcomes: Correct	CTRL Outcomes: Unavailable	CTRL Outcomes: Unsafe	CTRL Outcomes: Correct
LogCabin	Possible	2401	truncate	0	2355	46	0	0	2401
			crash	2355	0	46	0	0	2401
	Not possible	1695	truncate	0	1695	0	1695	0	0
			crash	1695	0	0	1695	0	0
ZooKeeper	Possible	2401	truncate	0	2355	46	0	0	2401
			crash	2355	0	46	0	0	2401
	Not possible	1695	truncate	0	1695	0	1695	0	0
			crash	1695	0	0	1695	0	0

Table 3.4: **Targeted Corruptions.** *The table shows results for targeted corruptions in log; we trigger two policies (truncate and crash) in the original systems. Recovery is possible when at least one intact copy exists; recovery is not possible when no intact copy exists.*

exists). In the remaining 1695 cases, recovery is not possible because one or more entries are corrupted on *all* the nodes. We inject targeted corruptions into two different sets of on-disk structures. In the first set, on a corruption, the original systems invoke the *truncate* action (i.e., they truncate faulty data and continue). In the second set, the original systems invoke the *crash* action (i.e., node crashes on detection). For example, while ZooKeeper *truncates* when the tail of a transaction is corrupted, it *crashes* the node if the transaction header is corrupted. CTRL always recovers the corrupted data from other replicas.

Table 3.4 shows the results. When recovery is possible, the original systems recover only in 46/2401 cases. In those 46 cases, no node or only

one node is corrupted. In the remaining 2355 cases, the original systems are either unsafe (for *truncate*) or unavailable (for *crash*). In contrast, CTRL correctly recovers in all 2401 cases. When a recovery is not possible (all copies corrupted), the original systems are either unsafe or unavailable in all cases. CTRL, by design, correctly remains unavailable since continuing would violate safety.

Random Block Corruptions and Errors. We initialize the cluster by replicating a few entries to all nodes. We first choose a random set of nodes. In each such node, we then corrupt a randomly selected file-system block (from the files implementing the log). We repeat this process, producing 5000 test cases. We similarly inject block errors. Since we inject a fault into a block, several entries and their checksums within the block will be faulty.

Table 3.5(a) shows the results. For *block corruptions*, original LogCabin is unsafe or unavailable in about 30% ($(738 + 793)/5000$) of cases. Similarly, original ZooKeeper is incorrect in about 30% of cases. On a *block error*, original LogCabin and ZooKeeper simply crash the node, leading to unavailability in about 50% of cases. In contrast, CTRL correctly recovers in all cases.

Faults with Crashed and Lagging Nodes. In the previous experiments, all entries were committed and present on all nodes. In this experiment, we inject crashes at different points on a random set of nodes while inserting entries. Thus, in the resultant log states, nodes could be lagging, entries could be uncommitted, and have different epochs on different nodes for the same log index.

$\langle S_1 : [a^1, _, _], S_2 : [b^2, c^3, _], S_3 : [b^2, _, _]\rangle$ is an example state where S_1 appends a at index 1 in epoch 1 (shown in superscript) and crashes, S_2 appends b at index 1 in epoch 2, replicates to S_3, then S_2, S_3 crash and

System	Experiment	Total Test Cases	Outcomes: Original Unavailable	Original Unsafe	Original Correct	CTRL Unavailable	CTRL Unsafe	CTRL Correct
LogCabin	Corruptions	5000	738	793	3469	0	0	5000
	Errors	5000	2497	0	2503	0	0	5000
ZooKeeper	Corruptions	5000	807	656	3537	0	0	5000
	Errors	5000	2469	0	2531	0	0	5000

(a) Random Block Corruptions and Errors.

System	Total Test Cases	Outcomes: Original Unavailable	Original Unsafe	Original Correct	CTRL Unavailable	CTRL Unsafe	CTRL Correct
LogCabin	5000	4194	141	665	0	0	5000
ZooKeeper	5000	1306	1806	1888	0	0	5000

(b) Corruptions with Lagging Nodes

Table 3.5: **Log Recovery.** *(a) shows results for random block corruptions and errors in the log. (b) shows results for random corruptions in the log with crashed and lagging nodes.*

recover, S_2 appends c in epoch 3 and crashes. From each such state, we corrupt different entries, generating 5000 test cases. For example, from the above state, we corrupt a on S1 and b, c on S_2. If S_2 is elected the

leader, S_2 needs to fix b from S_3 (since b is committed), discard c (c is uncommitted and cannot be recovered), and also instruct S1 to discard a (a is uncommitted) and replicate correct entry b. As shown in Table 3.5(b), CTRL correctly recovers from all such cases, while the original versions are unsafe or unavailable in many cases.

Model Checking. We also model checked CTRL's log recovery since it involves many corner cases, using a python-based model that we developed. We explored over 2.5M log states all of in which CTRL correctly recovered. Also, when key decisions are tweaked, the checker finds a violation immediately: for example, the leader concludes that a faulty entry is uncommitted only after gathering $\lfloor n/2 \rfloor + 1$ *dontHave* responses; if this number is reduced, then the checker finds a safety violation. We have also added the specification of CTRL's log recovery to the TLA+ specification of Raft [74] and confirmed that it correctly recovers from corruptions, while the original specification violates safety.

Snapshot Recovery

In this experiment, we insert a few entries and trigger the nodes to take a snapshot. We crash the nodes at different points, producing three possible states for each node: l, t, and g, where l is a state where the node has only the log (it has not taken a snapshot), t is a snapshot for which garbage collection has not been performed yet, and g is a snapshot which has been garbage collected. We produce all possible combinations of states across three nodes. On each such state, we randomly pick a set of nodes to inject faults, and corrupt a random combination of snapshots and log entries, generating 1000 test cases. For example, $\langle S_1 : t, S_2 : g, S_3 : l \rangle$ is a base state on which we corrupt snapshot t and a few preceding log entries on S_1 and g on S_2. In such a state, if S_1 becomes the leader, it has to fix its log from S_3, then has to locally recover its t snapshot, after which it has to fix

System	Total Test Cases	Outcomes					
		Original			CTRL		
		Unavailable	Unsafe	Correct	Unavailable	Unsafe	Correct
LogCabin	1000	297	257	446	0	0	1000
ZooKeeper	1000	417	200	383	0	0	1000

(a) Snapshot Recovery

System	Total Test Cases	Outcomes					
		Original			CTRL		
		Unavailable	Unsafe	Correct	Unavailable	Unsafe	Correct
LogCabin	1000	405	36	559	434	0	566
ZooKeeper	1000	329	192	479	502	0	498

(b) FS Metadata Faults

Table 3.6: **Snapshot and FS Metadata Faults.** *(a) and (b) show how* CTRL *recovers from snapshot and FS metadata faults, respectively.*

g on S_2. S_1 also needs to install the snapshot on S_3.

As shown in Table 3.6(a), CTRL correctly recovers from all such cases. Original LogCabin is incorrect in about half of the cases because it obliviously loads faulty snapshots sometimes and crashes sometimes. Original ZooKeeper crashes the node if it is unable to locally construct the data from the snapshot and the log, leading to unavailability; unsafety results because a faulty log is truncated in some cases.

File-system Metadata Faults

To test how Ctrl recovers from file-system metadata faults, we corrupt file-system metadata structures (such as inodes and directory blocks) resulting in unopenable files, missing files, and files with fewer or more bytes. We inject such faults in a randomly chosen file on one or two nodes at a time, creating 1000 test cases. Table 3.6(b) shows the results. In some cases, the faulty nodes in original versions crash because of a failed deserialization or assertion. However, sometimes original LogCabin and ZooKeeper do not detect the fault and continue operating, violating safety in 36 and 192 cases, respectively. In contrast, Ctrl reliably crashes the node on a file-system metadata fault, preserving safety always.

3.4.2 Performance

We now compare the common-case performance of the Ctrl versions against the original versions. During writes, the entries are first written to the on-disk log; snapshots are taken periodically in the background. Both LogCabin and ZooKeeper batch several log entries to improve write throughput. In addition to the above steps, Ctrl writes an identifier for each log entry at the head of the log. First, we run a write-only workload that exposes the worst-case overheads (caused by the additional writes) introduced by Ctrl. The workload runs for 300 seconds, inserting entries each of size 1K. Numbers reported are the average over five runs.

Figure 3.8(a) and (c) show the throughput on an HDD for varying number of clients in LogCabin and ZooKeeper, respectively. Ctrl writes the identifiers in a physically separate location compared to that of the entries; this separation induces a seek on disks in the update path. However, the seek cost is amortized when more requests are batched; Ctrl has an overhead of 8%-10% for 32 clients on disks. Figure 3.8(b) and (d) show throughput on an SSD; Ctrl adds very minimal overhead on SSDs

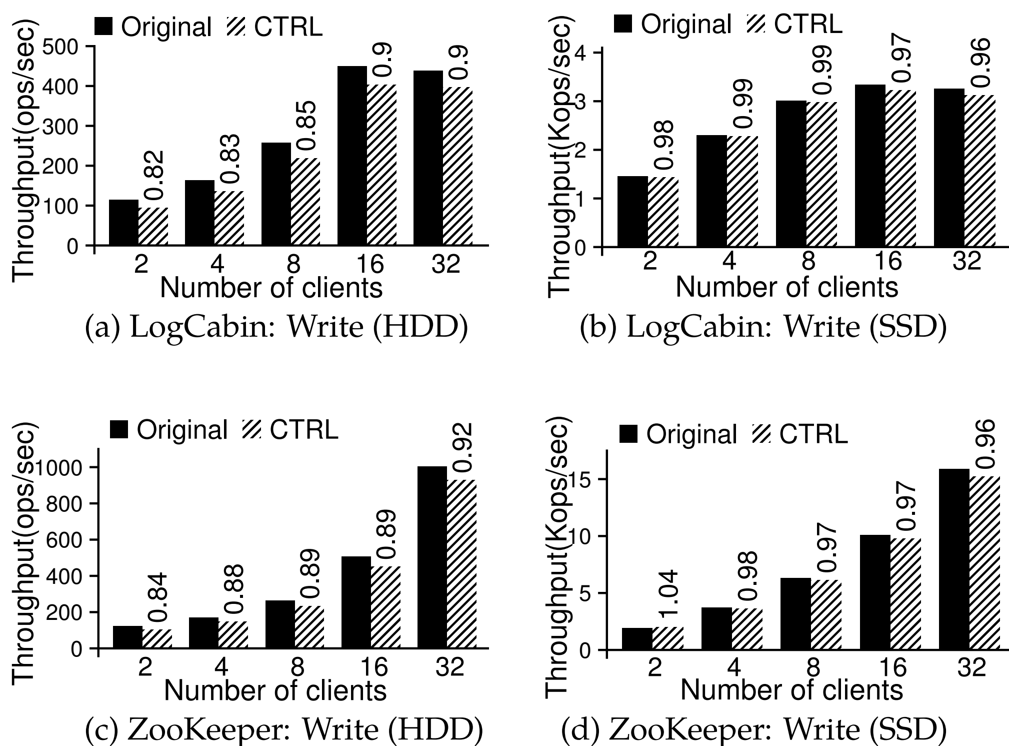
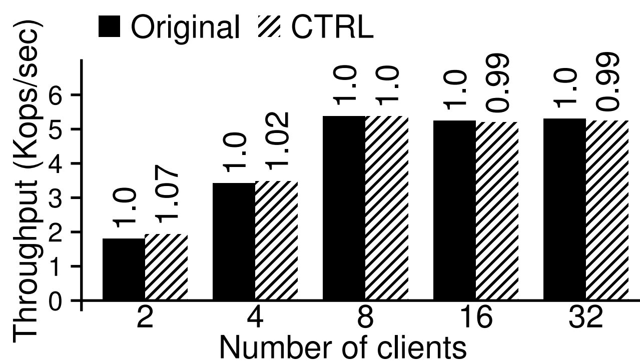


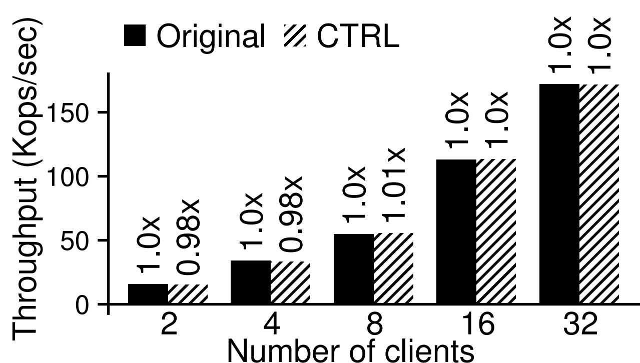
Figure 3.8: **Write Performance.** *(a) and (c) show the write throughput in original and* Ctrl *versions of LogCabin and ZooKeeper on an HDD. (b) and (d) show the same for SSD. The number on top of each bar shows the performance of* Ctrl *normalized to that of original.*

(4% in the worst case). Note that this workload performs only writes and therefore shows Ctrl's overheads in the worst case.

In both LogCabin and ZooKeeper, reads are served from memory; thus, the read paths should ideally not be affected by Ctrl. To confirm this, we run a read-only workload. Figure 3.9(a) and (b) show the throughput on an SSD for varying number of clients in LogCabin and ZooKeeper, respectively. As shown in the figure, Ctrl does not introduce any overheads and maintains the same performance as the original systems. We see similar results on HDDs (not shown in the figure).



(a) LogCabin: Read (SSD)



(b) ZooKeeper: Read (SSD)

Figure 3.9: **Read Performance.** *(a) and (b) show the read throughput in original and* Ctrl *versions of LogCabin and ZooKeeper on a SSD. The number on top of each bar shows the performance of* Ctrl *normalized to that of original.*

Fast Log Recovery. We now show how Ctrl can recover a faulty log quickly. To show the potential reduction in log-recovery time, we insert 30K log entries (each of size 1K) and corrupt the first entry on one node. In original LogCabin, the faulty node detects the corruption but truncates *all* entries; hence, the leader transfers all entries to bring the node up-to-date. Ctrl fixes only the faulty entry, reducing recovery time. The faulty node is fixed in 1.24 seconds (32MB transferred) in the original system, while Ctrl takes only 1.2 ms (7KB transferred). We see a similar reduction in log-recovery time in ZooKeeper.

3.5 Summary and Conclusions

Using redundancy to recover from failures is an important aspect of any truly reliable distributed system. However, despite this importance, even the most critical class of distributed storage systems do not effectively utilize the inherent data redundancy to recover from storage faults. Our analysis of existing approaches in the first part of this chapter revealed that most approaches do not use protocol-level knowledge to perform recovery, leading to safety violation or unavailability.

As a solution to this problem, in the second part, we introduced protocol-aware recovery (PAR), a new approach that exploits protocol-specific knowledge of the underlying distributed system to correctly recover from storage faults. We designed CTRL, a protocol-aware recovery approach for RSM systems. CTRL consists of a local storage layer and distributed recovery protocol; while the storage layer reliably detects storage faults, the distributed protocol recovers faulty data from redundant copies on other servers.

We implemented CTRL in two systems (LogCabin and ZooKeeper) that are based on two different consensus protocols. Through rigorous experiments, we showed that CTRL correctly recovers from a range of storage faults. We also showed that the reliability improvements of CTRL come with little to no performance overheads in the common case.

More broadly, we believe our work in this chapter is only a first step in hardening distributed systems to storage faults: while we have successfully applied the PAR approach to RSM systems, other classes of systems (e.g., primary-backup, Dynamo-style quorums) still remain to be analyzed. Recent related studies [92] have shown that these classes of distributed systems are vulnerable to storage faults too; experimental data has shown that these systems can silently return corrupted data, lose data, or become unavailable. We believe the PAR approach can also be applied to these systems to improve their resiliency to storage faults. Further, new

storage technologies such as non-volatile memory (NVM) and QLC Nand devices are finding rapid adoption in data centers. However, recent studies have shown that these new technologies also suffer from media corruption and errors [220, 235]. Thus, we believe the ideas presented in this chapter can help improve the resiliency of many future deployments.

4
Situation-Aware Updates and Crash Recovery

In this chapter, we analyze the resiliency to crash failures and performance characteristics of different replication protocols (such as Paxos [130], Viewstamped Replication [140], Raft [175], and ZAB [119]) used by distributed systems. We introduce *situation-aware updates and crash recovery* (SAUCR), a new approach to replication in a distributed system.

In the first part of this chapter, we show that a dichotomy exists with respect to how and where current approaches store system state. In the disk-durable approach, critical state is replicated to persistent storage, while in the memory-durable approach, data is replicated only to the (volatile) memory. Our analysis shows that neither of these approaches is ideal. Disk-durable approaches offer strong reliability but poor performance; memory-durable approaches, in contrast, deliver high performance but provide poor durability and availability.

Thus, in the second part of this chapter, we introduce *situation-aware updates and crash recovery* (SAUCR), a new approach to performing replicated data updates in a distributed system. SAUCR adapts the update protocol to the current situation: with many nodes up, SAUCR buffers updates in memory; when failures arise, SAUCR flushes updates to disk. This situation-awareness enables SAUCR to achieve high performance while offering strong durability and availability guarantees. We implement a pro-

totype of Saucr in ZooKeeper. Through rigorous crash testing, we demonstrate that Saucr significantly improves durability and availability compared to systems that always write only to memory. We also show that Saucr's reliability improvements come at little or no cost: Saucr's overheads are within 0%-9% of memory-durable ZooKeeper. This chapter is based on the paper, *Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems*, published in OSDI 18 [11].

We first present our study of existing approaches to replication and describe how crash failures arise in modern data-center environments (§4.1). We then describe the design of Saucr (§4.2). Next, we describe Saucr's prototype implementation (§4.3) and present our evaluation (§4.4). We then discuss concerns related to Saucr's adoption in practice (§4.5). Finally, we summarize and conclude (§4.6).

4.1 Existing Approaches to Replication

In this section, we first describe disk-durable and memory-durable protocols, the common protocols used by most distributed systems; we explain how these protocols exhibit undesirable properties such as poor performance, durability, or availability. Next, more importantly, we draw attention to how these protocols are *static* in nature: how they always update and recover in a constant way, without adapting to failures in the system, resulting in said undesirable properties.

4.1.1 Disk-Durable Protocols

Disk-durable protocols always update the disk on a certain number of nodes upon every data modification. For example, ZooKeeper [14], etcd [64], and other systems [40, 144, 167, 197] persist every update on a majority of nodes before acknowledging clients.

	Mode	Avg. Latency (μs)	Throughput (ops/s)
HDD cluster-1	fsync-S disabled	327.86	3050.1
	disk durability	16665.18 (50.8× ↑)	60.0 (50.8× ↓)
SSD cluster-2	fsync-S disabled	461.2	2168.34
	disk durability	1027.3 (2.3× ↑)	973.4 (2.3× ↓)

Table 4.1: **Disk Durability Overheads.** *The table shows the overheads of disk durability. The experimental setup is detailed in §4.4.2.*

With the exception of subtle bugs [12], disk-durable protocols offer excellent durability and availability. Specifically, committed data will never be lost under any crash failures. Further, as long as a majority of nodes are functional, the system will remain available. Unfortunately, such strong durability and availability guarantees come at a cost: poor performance.

Disk-durable protocols operate with caution and pessimistically flush updates to the disk (e.g., by invoking the `fsync` system call [29, 181]). Such forced writes in the critical path are expensive, often prohibitively so. To highlight these overheads, we conduct a simple experiment with ZooKeeper in the following modes: first, in the disk-durable configuration in which the `fsync` calls are enabled; second, with `fsync` calls disabled. A client sends update requests in a closed loop to the leader which then forwards the requests to the followers. We run the experiment on a five-node cluster and thus at least three servers must persist the data before acknowledgment.

As shown in Table 4.1, on HDDs, forced writes incur a 50× performance overhead compared to the `fsync`-disabled mode. Even on SSDs, the cost of forced writes is high (2.3×). While batching across many clients may alleviate some overheads, disk-durable protocols are fundamentally

limited by the cost of forced writes and thus suffer from high latencies and low throughput.

A disk-durable update protocol is usually accompanied by a disk-based recovery protocol. During crash-recovery, a node can immediately join the cluster just after it recovers the data from its disk. A recovering node can completely trust its disk because the node would not have acknowledged any external entity before persisting the data. However, the node may be lagging: it may not contain some data that other nodes might have stored after it crashed. Even in such cases, the node can immediately join the cluster; if the node runs for an election, the leader-election protocol will preclude this node from becoming the leader because it has not stored some data that the other nodes have [12, 175]. If a leader already exists, the node fetches the missed updates from the leader.

4.1.2 Memory-Durable Protocols

Given the high overheads imposed by a disk-durable protocol, researchers and practitioners alike have proposed alternative protocols [50, 171], in which the data is always buffered in memory, achieving good performance. We call such protocols *memory-durable* protocols.

Oblivious Memory Durability

The easiest way to achieve memory "durability" is *oblivious memory durability*, in which any forced writes in the protocol are simply disabled, unaware of the risks of only buffering the data in memory. Most systems provide such a configuration option [21, 60, 76, 197]; for example, in ZooKeeper, turning off the *forceSync* flag disables all `fsync` calls [16]. Turning off forced writes increases performance significantly, which has tempted practitioners to do so in many real-world deployments [85, 115, 178].

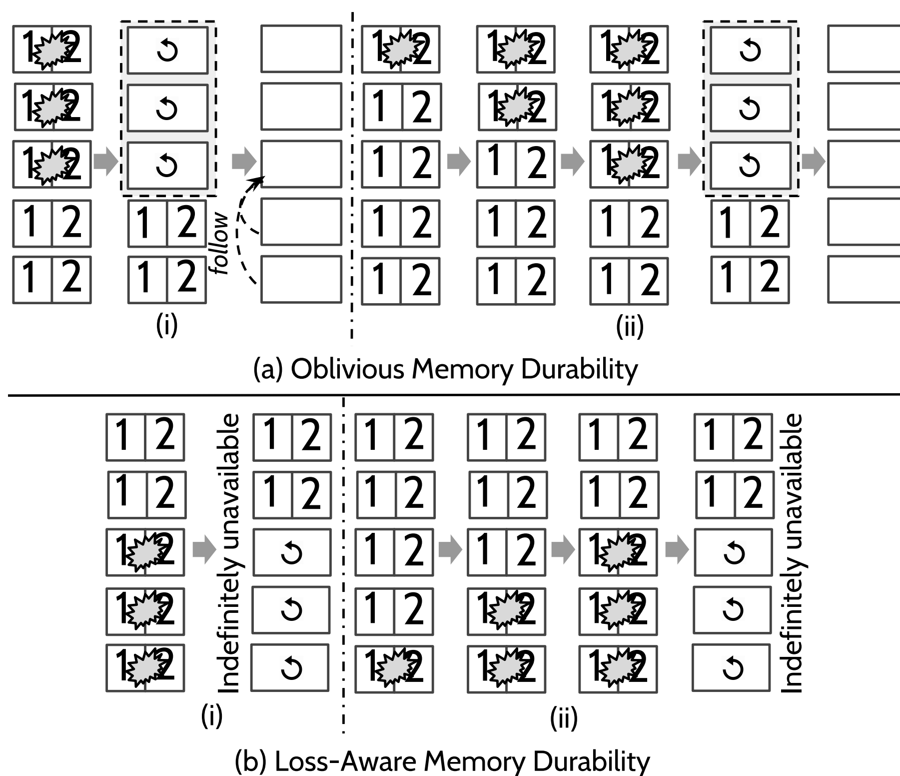


Figure 4.1: **Problems in Memory-Durable Approaches.** *(a) and (b) show how a data loss or an unavailability can occur with oblivious and loss-aware memory durability, respectively. In (i), the nodes fail simultaneously; in (ii), they fail non-simultaneously, one after the other.*

Unfortunately, disabling forced writes might lead to a data loss [16, 126] or sometimes even an unexpected data corruption [252]. Developers and practitioners have reported several instances where this unsafe practice has led to disastrous data-loss events in the real world [19, 86].

Consider the scenarios shown in Figure 4.1(a), in which ZooKeeper runs with *forceSync* disabled. If a majority of nodes crash and recover, data could be silently lost. Specifically, the nodes that crash could form a majority and elect a leader among themselves after recovery; however, this majority of nodes have lost their volatile state and thus do not know

of the previously committed data, causing a silent data loss. The intact copies of data on other nodes (servers 4 and 5) can be overwritten by the new leader because the followers always follow the leader's state in ZooKeeper [12, 175].

Loss-Aware Memory Durability

Given that naïvely disabling forced writes may lead to a silent data loss, researchers have examined more careful approaches. In these approaches, a node, after a crash and a subsequent reboot, realizes that it might have lost its data; thus, a recovering node first runs a distinct recovery protocol. We call such approaches *loss-aware memory-durable* approaches.

The view-stamped replication (VR) protocol [171] is an example of this approach. Similarly, researchers at Google observed that they could optimize their Paxos-based system [50] by removing disk flushes, given that the nodes run a recovery protocol. For simplicity, we use only VR as an example for further discussion.

In VR, when a node recovers from a crash, it first marks itself to be in a *recovering* state, in which the node can neither participate in replication nor give votes to elect a new leader (i.e., a view change) [140]. Then, the node sends a recovery message to other servers. A node can respond to this message if it is *not* in the *recovering* state; the responding node sends its data to the recovering node. Once the node collects responses from a majority of servers (including the leader of the latest view), it can fix its data. By running a recovery protocol, this approach prevents a silent data loss.

Unfortunately, the loss-aware approach can lead to unavailability in many failure scenarios. Such an unavailability event could be *permanent*: the system may remain unavailable indefinitely even after all nodes have recovered from failures. For example, in Figure 4.1(b), a majority of nodes crash and recover. However, after recovering from the crash, none of the

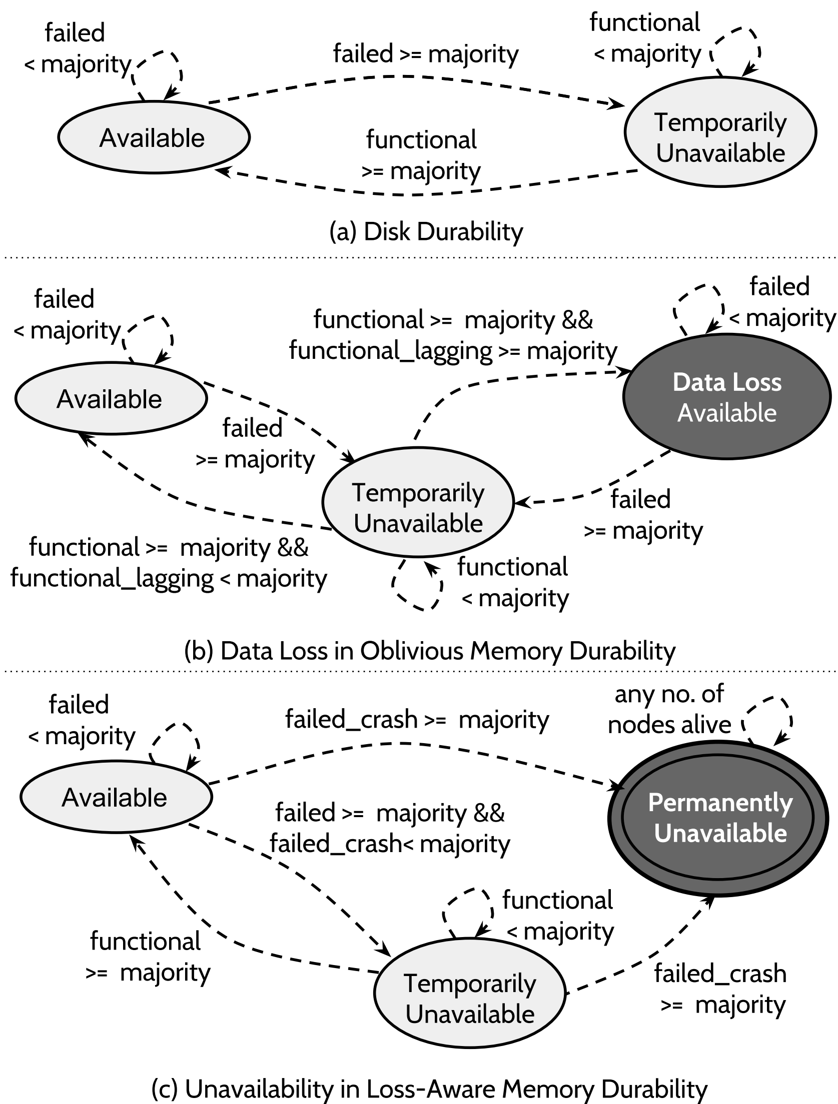


Figure 4.2: **Summary of Protocol Behaviors and Guarantees.** *The figure shows how the disk-durable and memory-durable protocols behave under failures and the guarantees they provide.*

nodes will be able to collect recovery responses from a majority of nodes (because nodes in the *recovering* state cannot respond to the recovery messages), leading to permanent unavailability.

Protocols Summary. Figure 4.2 summarizes the behaviors of the disk-durable and memory-durable protocols. A node either could be functional or could have failed (crashed or partitioned). Disk-durable protocols remain available as long as a majority are functional. The system becomes *temporarily* unavailable if a majority fail; however, it becomes available once a majority recover. Further, the protocol is durable at all times.

The oblivious memory-durable protocol becomes temporarily unavailable if a majority fail. After recovering from a failure, a node could be lagging: it either recovers from a crash, losing all its data, or it recovers from a partition failure, and so it may not have seen updates. If such functional but lagging nodes form a majority, the system can silently lose data.

The loss-aware memory-durable approach becomes temporarily unavailable if the system is unable to form a majority due to partitions. However, the system becomes permanently unavailable if a majority or more nodes crash at any point; the system cannot recover from such a state, regardless of how many nodes recover.

4.1.3 Failure Patterns in Data Centers

Existing approaches to replication present an unsavory tradeoff: they provide either reliability or performance, but not both. An ideal distributed replication protocol must deliver high performance while providing strong resiliency to crash failures. Such a design needs a careful understanding of how failures occur in modern data-center environments, which we discuss next.

Sometimes, node failures are *independent*. For example, in large deployments, single-node failure events are often independent: a crash of one node (e.g., due to a power failure) does not affect some other node. It is unlikely for many such independent failures to occur together, especially given the use of strategies such as failure-domain-aware place-

ment [13, 127, 156]. Under such a condition, designing a protocol that provides both high performance and strong guarantees is fairly straightforward: the protocol can simply buffer updates in memory always (similar to existing memory-durable approaches). Given that a majority will not be down at any point, the system will always remain available. Further, at least one node in the alive majority will contain all the committed data, preventing a data loss.

Unfortunately, in reality, such a failure-independence assumption is rarely justified. In many deployments, failures can be correlated [36, 58, 68, 106, 216, 233]. During such correlated crashes, several nodes fail together, often due to the same underlying cause such as rolling reboots [88], bad updates [169], bad inputs [71], or data-center-wide power outages [124].

Given that failures can be correlated, it is likely that the above naïve protocol may lose data or become unavailable. An ideal protocol must provide good performance and strong guarantees in the presence of correlated failures. However, designing such a protocol is challenging. At a high level, if the updates are buffered in memory (aiming to achieve good performance), a correlated failure may take down all the nodes together, causing the nodes to lose the data, affecting durability.

Although many or all nodes fail together, a correlated failure does not mean that the nodes fail at the same instant; the nodes can fail either non-simultaneously or simultaneously. With non-simultaneous correlated crashes, a time gap between the individual node failures exists. For instance, a popular correlated crash scenario arises due to bad inputs: many nodes process a bad input and crash together [71]. However, such a bad input is not applied at exactly the same time on all the nodes (for instance, a leader applies an input before its followers), causing the individual failures to be non-simultaneous.

In contrast, with simultaneous correlated crashes, such a window between failures does not exist; all nodes may fail before any node can detect

a failure and react to it. However, we conjecture that such truly simultaneous crashes are extremely rare; we call this the Non-Simultaneity Conjecture (NSC). Publicly available data supports NSC. For example, a study of failures in Google data centers [88] showed that in most correlated failures, nodes fail one after the other, usually a few seconds apart.

We also analyze the time gap between failures in the publicly available Google cluster data set [96]. This data set contains traces of machine events (such as the times of node failures and restarts) of about 12K machines over 29 days and contains about 10K failure events. From the traces, we randomly pick five machines (without considering failure domains) and examine the timestamps of their failures. We repeat this 1M times (choosing different sets of machines). We find that the time between two failures among the picked machines is greater than 50 ms in 99.9999% of the cases. However, we believe the above percentage is a conservative estimate, given that we did not pick the machines across failure domains; doing so is likely to increase the time between machine failures. Thus, we observe that truly simultaneous machine failures are rare: a gap of 50 ms or more almost always exists between the individual failures.

Given that in most (if not all) failure scenarios, a window of time exists between the individual failures, a replication protocol can take advantage of the window to react and perform a preventive measure (e.g., flushing to disk). A system that exploits this asynchrony in failures can improve durability and availability significantly. However, as we discuss next, none of the existing replication protocols exploit this opportunity.

4.1.4 Non-Reactiveness and Static Nature

We observe that existing update protocols do not react to failures. While it may be difficult to react to truly simultaneous failures (that are rare), with independent and non-simultaneous failures (which is far more prevalent), an opportunity exists to detect failures and perform a corrective

step. However, existing protocols do not react to *any* failure.

For example, the oblivious memory-durable protocol can lose data, regardless of the simultaneity of the failures. Specifically, a data loss occurs both in Figure 4.1(a)(i) in which the nodes crash simultaneously and (a)(ii) in which they fail non-simultaneously. Similarly, the loss-aware approach can become unavailable, regardless of the simultaneity of the failures (as shown in Figure 4.1(b)). This is the reason we do not differentiate simultaneous and non-simultaneous failures in Figure 4.2; the protocols behave the same under both failures.

Next, we note that the protocols are *static* in nature: they always update and recover in a constant way, regardless of the situation; this situation-obliviousness is the cause for poor performance or reliability. For example, the disk-durable protocol constantly anticipates failures, forcing writes to disk even when nodes never or rarely crash; this unnecessary pessimism leads to poor performance. In contrast, when nodes rarely crash, a situation-aware approach would buffer updates in memory, achieving high performance. Similarly, the memory-durable protocol always optimistically buffers updates in memory even when only a bare majority are currently functional; this unwarranted optimism results in poor durability or availability. In contrast, when only a bare majority are alive, a situation-aware approach would safely flush updates to disk, improving durability and availability.

4.1.5 Summary: The Need for a Situation-Aware Approach

In this section, we discussed how existing replication schemes offer either strong reliability or high performance. We showed that this tradeoff results from the static nature of existing protocols; current approaches do not adapt to failures and statically fix how updates will be committed. We discussed how failures typically arise in data-center environments and

showed how, in almost all cases, a gap exists between individual failures. However, existing replication protocols do not take advantage of this window to react to failures.

To resolve the tension between strong reliability and performance, in the next section, we introduce a new approach, *situation-aware updates and crash recovery* or SAUCR. SAUCR reacts to failures quickly with corrective measures and adapts to the current situation of the system. Such reactiveness and situation-awareness enable SAUCR to achieve high performance similar to a memory-durable protocol while providing strong guarantees similar to a disk-durable protocol.

4.2 Situation-Aware Updates and Recovery

Our analysis in the previous section revealed that existing approaches to replication offer either reliability or performance, but not both. The key reason why existing approaches compromise on reliability or performance is that they statically fix where and how to commit updates. More specifically, disk-durable protocols always write data to disk, and do not adapt to current conditions, leading to poor performance. Similarly, memory-durable protocols always buffer updates in memory, and do not react to failures, leading to poor reliability.

In contrast, a replication protocol that can react to failures and adapt itself to the current situation, can obtain both strong reliability and high performance. Such a dynamic, situation-aware approach would operate in the memory-durable mode when failures are rare and would switch to disk-durable mode if and when failures arise. However, to realize this goal, a protocol must have enough time to detect and react to failures as they are occurring. Fortunately, as we discussed in the previous section, in most (if not all) failure scenarios, a window of time exists between the individual failures.

We showed that with independent failures, such a time gap between failures obviously exists. Even when failures are correlated (where many nodes can fail together), the nodes do not necessarily fail at the same instant: the nodes fail non-simultaneously in most cases. With non-simultaneous correlated failures, a time gap (ranging from a few milliseconds to a few seconds) exists between the individual failures. With simultaneous failures, in contrast, such a window does not exist. However, we described how such truly simultaneous failures are extremely rare by putting forth the Non-Simultaneity Conjecture (NSC).

The above observation forms the key idea of SAUCR, which is the mode of replication should depend upon the situation the distributed system is in at a given time. In the common case, with many (or all) nodes up and running, SAUCR runs in memory-durable mode, thus achieving excellent throughput and low latency; when nodes crash or become partitioned, SAUCR transitions to disk-durable operation, thus ensuring safety at a lower performance level. Given that a window of time exist between failures in almost all failure scenarios, SAUCR can safely move from its fast mode to its slow-and-safe mode.

In the common case, SAUCR operates in the fast mode. When failures arise, SAUCR quickly detects them and performs two corrective measures. First, the nodes flush their data to disk, preventing an imminent data loss or unavailability. Second, SAUCR commits subsequent updates in slow mode, in which the nodes synchronously write to disk, sacrificing performance to improve reliability. When a node recovers from a crash, it performs mode-aware recovery. The node recovers its data either from its local disk or from other nodes depending on whether it operated in slow or fast mode before it crashed.

In the remainder of this section, we first describe the failure model that SAUCR intends to tolerate (§4.2.1) and outline its guarantees (§4.2.2). We then explain SAUCR's modes of operation (§4.2.3), failure reaction (§4.2.4),

and crash recovery (§4.2.5, §4.2.6). Then, we summarize Saucr's key aspects and describe the guarantees in detail (§4.2.7).

4.2.1 Failure Model

Most distributed systems intend to tolerate only fail-recover failures. Similar to these systems, Saucr also intends to only handle fail-recover failures [98, 111, 130, 175] and not Byzantine failures [48, 131]. In the fail-recover model, nodes may fail any time and recover later. For instance, a node may crash due to a power loss and recover when the power is restored. When a node recovers, it loses all its volatile state and is left only with its on-disk data. We assume that persistent storage will be accessible after recovering from the crash and that it will not be corrupted [92]. In addition to crashing, sometimes, a node could be partitioned and may later be able to communicate with the other nodes; however, during such partition failures, the node does not lose its volatile state.

4.2.2 Guarantees

We consider three kinds of failures: independent, correlated non-simultaneous, and correlated simultaneous failures. Saucr can tolerate any number of independent and non-simultaneous crashes; under such failures, Saucr always guarantees durability. As long as a majority of servers eventually recover, Saucr guarantees availability. Under simultaneous correlated failures, if a majority or fewer nodes crash, and if eventually a majority recover, Saucr will provide durability and availability. However, if *more than* a majority crash simultaneously, then Saucr cannot guarantee durability and so will remain unavailable. However, we believe such truly simultaneous crashes are extremely rare. We discuss the guarantees in more detail later (§4.2.7).

4.2.3 SAUCR Modes

We first describe some properties common to many majority-based systems. We then highlight how SAUCR differs from existing systems in key aspects.

Most majority-based systems are *leader-based* [18, 175]; the clients send updates to the leader which then forwards them to the followers. The updates are first stored in a log and are later applied to an application-specific data structure. A leader is associated with an *epoch*: a slice of time; for any given epoch, there could be at most one leader [18, 175]. Because only the leader proposes an update, each update is uniquely qualified by the epoch in which the leader proposed it and the index of the update in the log. The leader periodically checks if a follower is alive or not via heartbeats. If the followers suspect that the leader has failed, they compete to become the new leader in a new epoch. Most systems guarantee the *leader-completeness* property: a candidate can become the leader only if it has stored all items that have been acknowledged as committed [12, 175]. SAUCR retains all the above properties of majority-based systems.

In a memory-durable system, the nodes always buffer updates in memory; similarly, the updates are always synchronously persisted in a disk-durable system. SAUCR changes this fundamental attribute: SAUCR either buffers the updates or synchronously flushes them to disk depending on the situation. When more nodes than a bare minimum to complete an update are functional, losing those additional nodes will not result in an immediate data loss or unavailability; in such situations, SAUCR operates in fast mode. Specifically, SAUCR operates in fast mode if *more than* a bare majority are functional (i.e., functional $> \lfloor n/2 \rfloor + 1$, where n is the total nodes, typically a small odd number). If nodes fail and only a bare majority ($\lfloor n/2 \rfloor + 1$) are functional, losing even one additional node may lead to a data loss or unavailability; in such situations, SAUCR switches to the

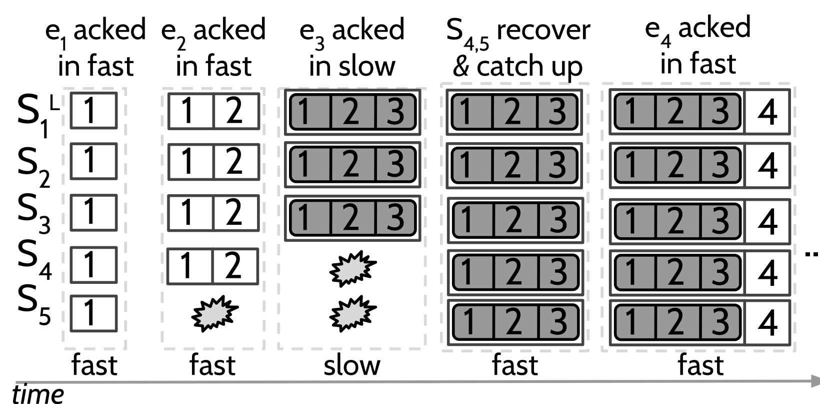


Figure 4.3: **Saucr Modes.** *The figure shows how* Saucr*'s modes work.* S_1 *is the leader. Entries in a white box are committed but are only buffered (e.g.,* e_1 *and* e_2 *in the first and second states). Entries shown grey denote that they are persisted (e.g.,* $e_1 - e_3$ *in the third state). In fast mode, a node loses its data upon a crash and is annotated with a crash symbol (e.g.,* S_5 *has lost its data in the second state).*

slow mode. Because the leader continually learns about the status of the followers, the leader determines the mode in which a particular request must be committed.

We use Figure 4.3 to give an intuition about how Saucr's modes work. At first, all the nodes are functional and hence the leader S_1 replicates entry e_1 in fast mode. The followers acknowledge e_1 before persisting it (before invoking `fsync`); similarly, the leader also only buffers e_1 in memory. In fast mode, the leader acknowledges an update only after $\lfloor n/2 \rfloor + 2$ nodes have buffered the update. Because at least four nodes have buffered e_1, the leader acknowledges e_1 as committed. Now, S_5 crashes; the leader detects this but remains in fast mode and commits e_2 in fast mode.

Next, S_4 also crashes, leaving behind a bare majority; the leader now immediately initiates a switch to slow mode and replicates all subsequent entries in slow mode. Thus, e_3 is replicated in slow mode. Committing an entry in slow mode requires at least a bare majority to persist the entry

to their disks; hence, when e_3 is persisted on three nodes, it is committed. Further, the first entry persisted in slow mode also persists all previous entries buffered in memory; thus, when e_3 commits, e_1 and e_2 are also persisted. Meanwhile, S_4 and S_5 recover and catch up with other nodes; therefore, the leader switches back to fast mode, commits e_4 in fast mode, and continues to commit entries in fast mode until further failures.

4.2.4 Failure Reaction

In the common case, with all or many nodes alive, SAUCR operates in fast mode. When failures arise, the system needs to detect them and switch to slow mode or flush to disk. The basic mechanism SAUCR uses to detect failures is that of heartbeats.

Follower Failures and Mode Switches. If a follower fails, the leader detects it via missing heartbeats. If the leader suspects that only a bare majority (including self) are functional, it immediately initiates a switch to slow mode. The leader sends a special request (or uses an outstanding request such as e_3 in the above example) on which it sets a flag to indicate to the followers that they must respond only after persisting the request; this also ensures that all previously buffered data will be persisted. All subsequent requests are then replicated in slow mode. When in fast mode, the nodes periodically flush their buffers to disk in the background, without impacting the client-perceived performance. These background flushes reduce the amount of data that needs to be written when switching to slow mode. Once enough followers recover, the leader switches back to fast mode. To avoid fluctuations, the leader switches to fast mode after confirming a handful number of times that it promptly gets a response from more than a bare majority; however, a transition to slow mode is immediate: the first time the leader suspects that only a bare majority of nodes are alive.

Leader Failures and Flushes. The leader takes care of switching between

modes. However, the leader itself may fail at any time. The followers quickly detect a failed leader (via heartbeats) and flush all their buffered data to disk. Again, the periodic background flushes reduce the amount of data that needs to be written.

4.2.5 Enabling Safe Mode-Aware Recovery

When a node recovers from a crash, it may have lost some data if it had operated in fast mode; in this case, the node needs to recover its lost data from other nodes. In contrast, the node would have all the data it had logged on its disk if it had crashed in slow mode or if it had flushed after detecting a failure; in such cases, it recovers the data only from its disk. Therefore, a recovering node first needs to determine the mode in which it last operated. Moreover, if a node recovers from a fast-mode crash, the other nodes should maintain enough information about the recovering node. We now explain how Saucr satisfies these two requirements.

Persistent Mode Markers

The Saucr nodes determine their pre-crash mode as follows. When a node processes the first entry in fast mode, it synchronously persists the epoch-index pair of that entry to a structure called the *fast-switch-entry*. Note that this happens only for the first entry in the fast mode. In the slow mode or when flushing on failures, in addition to persisting the entries, the nodes also persist the epoch-index pair of the latest entry to a structure called the *latest-on-disk-entry*. To determine its pre-crash mode, a recovering node compares the above two on-disk structures. If its *fast-switch-entry* is ahead[1] of its *latest-on-disk-entry*, then the node concludes that it was in the fast mode. Conversely, if the *fast-switch-entry* is behind the

[1] An entry a is ahead of another entry b if (a.epoch $>$ b.epoch) or (a.epoch $==$ b.epoch and a.index $>$ b.index).

latest-on-disk-entry, then the node concludes that it was in the slow mode or it had safely flushed to disk.

Replicated LLE Maps

Once a node recovers from a crash, it must know how many entries it had logged in memory or disk before it crashed. We refer to this value as the *last logged entry* or LLE of that node. The LLE-recovery step is crucial because only if a node knows its LLE, it can participate in elections. Specifically, a candidate requests votes from other nodes by sending its LLE. A participant grants its vote to a candidate if the participant's LLE and current epoch are not ahead of the candidate's LLE and current epoch, respectively [175] (provided the participant had not already voted for another candidate in this epoch).

In a majority-based system, as long as a majority of nodes are alive, the system must be able to elect a leader and make progress [26, 175]. It is possible that the system only has a bare majority of nodes including the currently recovering node. Hence, it is crucial for a recovering node to immediately recover its LLE; if it does not, it cannot participate in an election or give its vote to other candidates, rendering the system unavailable.

If a node recovers from a slow-mode crash, it can recover its LLE from its disk. However, if a node recovers from a fast-mode crash, it would *not* have its LLE on its disk; in this case, it has to recover its LLE from other nodes. To enable such a recovery, as part of the replication request, the leader sends a map of the last (potentially) logged entry of each node to every node. The leader constructs the map as follows: when replicating an entry at index i in epoch e, the leader sets the LLE of all the functional followers and self to e.i and retains the last successful value of LLE for the crashed or partitioned followers. For instance, if the leader (say, S_1) is replicating an entry at index 10 in epoch e to S_2, S_3, and S_4, and if S_5 has crashed after request 5, then the map will be $\langle S_1$:e.10, S_2:e.10, S_3:e.10,

S_4:e.10, S_5:e.5⟩. We call this map the *last-logged entry map* or LLE-MAP. In the fast mode, the nodes maintain the LLE-MAP in memory; in slow mode, the nodes persist the LLE-MAP to the disk.

4.2.6 Crash Recovery

In a disk-durable system, a node recovering from a crash performs three distinct recovery steps. First, it recovers its LLE from its disk. Second, it competes in an election with the recovered LLE. The node may either become the leader or a follower depending on its LLE's value. Third, the node recovers any missed updates from other nodes. If the node becomes the leader after the second step, it is guaranteed to have all the committed data because of the leader-completeness property [12, 175], skipping the third step. If the node becomes a follower, it might be lagging and so fetches the missed updates from the leader.

In SAUCR, a node recovering from a crash could have operated either in slow or fast mode before it crashed. If the node was in slow mode, then its recovery steps are identical to the disk-durable recovery described above; we thus do not discuss slow-mode crash recovery any further. A fast-mode crash recovery, however, is more involved. First, the recovering node would not have its LLE on its disk; it has to carefully recover its LLE from the replicated LLE-MAPs on other nodes. Second, it has to recover its lost data irrespective of whether it becomes the leader or a follower. We explain how a node performs the above crash-recovery steps.

Max-Among-Minority. A SAUCR node recovering from a fast-mode crash recovers its LLE using a procedure that we call *max-among-minority*. In this procedure, the node first marks itself to be in a state called *recovering* and then sends an LLE query to all other nodes. A node may respond to this query only if it is in a recovered (*not recovering*) state; if it is not, it simply ignores the query. Note that a node can be in the recovered state in two ways. First, it could have operated in fast mode and not crashed yet;

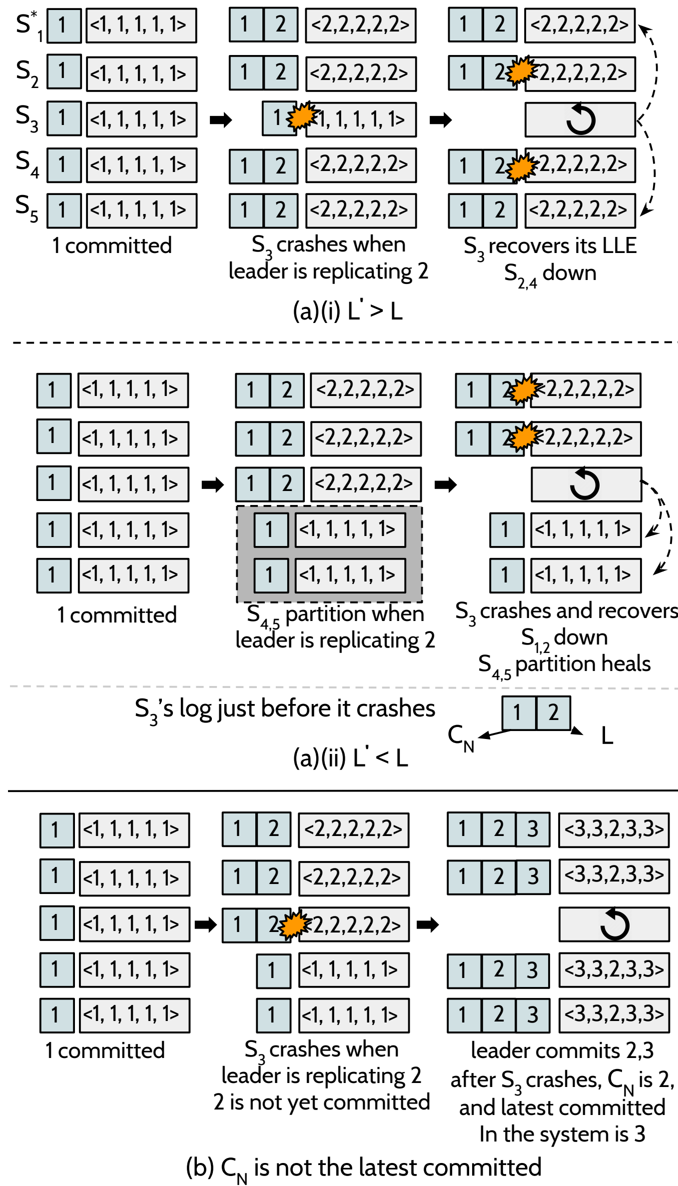


Figure 4.4: **LLE Recovery.** *The figure shows how L' may not be equal to L. For each node, we show its log and* `LLE-MAP`*. The leader (S_1) is marked *; crashed nodes are annotated with a crash symbol; nodes partitioned are shown in a dotted box; epochs are not shown.*

second, it could have last operated in slow mode and so has the LLE-MAP on its disk. The recovering node waits to get responses from at least a bare minority of nodes, where $\mathtt{bare\text{-}minority} = \lfloor n/2 \rfloor$; once the node receives a bare-minority responses, it picks the maximum among the responses as its LLE. Finally, the node recovers the actual data up to the recovered LLE. For now, we assume that at least a bare minority will be in recovered state; we soon discuss cases where only fewer than a bare minority are in the recovered state (§4.2.7).

We argue that the max-among-minority procedure guarantees safety, i.e., it does not cause a data loss. To do so, let us consider a node N that is recovering from a fast-mode crash and let its actual last-logged entry (LLE) be L. When N runs the max-among-minority procedure, it retrieves L' as its LLE and recovers all entries up to L'.

If N recovers exactly all entries that it logged before crashing (i.e., L'=L), then it is as though N had all the entries on its local disk (similar to how a node would recover in a disk-durable protocol, which is safe). Therefore, if the retrieved L' is equal to the actual last-logged entry L, the system would be safe.

However, in reality, it may not be possible for N to retrieve an L' that is exactly L. If N crashes after the leader sends a replication request but before N receives it, N may retrieve an L' that is greater than L. For example, consider the case shown in Figure 4.4(a)(i). The leader (S_1) has successfully committed entry-1 in fast mode and now intends to replicate entry-2; hence, the leader populates the LLE-MAP with 2 as the value for all the nodes. However, S_3 crashes before it receives entry-2; consequently, its LLE is 1 when it crashed. However, when S_3 recovers its LLE from LLE-MAPs of S_1 and S_5 using the max-among-minority algorithm, the recovered L' will be 2 which is greater than 1. Note that if L' is greater than L, it means that N will recover additional entries that were not present in its log, which is safe. Similarly, it is possible for N to retrieve an L' that

is smaller than L. For instance, in Figure 4.4(a)(ii), S_3 has actually logged two entries; however, when it recovers, its L' will be 1 which is smaller than the actual LLE 2. $L' < L$ is the only case that needs careful handling.

We now show that the system is safe even when the recovered L' is smaller than L. We first establish a lower bound for L' that guarantees safety. Then, we show that max-among-minority ensures that the recovered L' is at least as high as the established lower bound.

Lower bound for L'. Let N's log when it crashed be D and let C_N be the last entry in D that is committed. For example, in Figure 4.4(a)(ii), for S_3, D contains entries 1 and 2, and the last entry in D that was committed is 1. Note that C_N need not be the latest committed entry; the system might have committed more entries after N crashed but none of these entries will be present in N's log. For example, in Figure 4.4(b), for S_3, C_N is 2 while the latest committed entry in the system is 3.

For the system to be safe, all *committed* entries must be recovered, while the *uncommitted* entries need *not* be recovered. For example, in Figure 4.4(a)(ii), it is safe if S_3 does not recover entry-2 because entry-2 is uncommitted. However, it is unsafe if N does not recover entry-1 because entry-1 is committed. For instance, imagine that S_3 runs an incorrect recovery algorithm that does not recover entry-1 in Figure 4.4(a)(ii). Now, if S_1 and S_2 also run the incorrect algorithm, then it is possible for S_1, S_2, and S_3 to form a majority and lose committed entry-1. Therefore, if the recovery ensures that N recovers all the entries up to C_N, committed data will not be lost, i.e., L' must be at least as high as the last entry in N's log that is committed. In short, the lower bound for L' is C_N. Next, we show that indeed the L' recovered by max-among-minority is equal to or greater than C_N.

Proof Sketch for $L' \geqslant C_N$. We prove by contradiction. Consider a node N that is recovering from a fast-mode crash and that C_N is the last entry in N's log that was committed. During recovery, N queries a bare minority.

Let us suppose that N recovers an L' that is less than C_N. This condition can arise if a bare minority of nodes hold an LLE of N in their LLE-MAPs that is less than C_N. This is possible if the bare minority crashed long ago and recently recovered, or they were partitioned. However, if a bare minority had crashed or partitioned, it is *not* possible for the remaining bare majority to have committed C_N in fast mode (recall that a fast-mode commitment requires at least $\mathtt{bare\text{-}majority} + 1$ nodes to have bufferred C_N and updated their LLE-MAPs). Therefore, C_N could have either been committed only in slow mode or not committed at all. However, if C_N was committed in slow mode, then N would be recovering from a slow-mode crash which contradicts the fact that N is recovering from a fast-mode crash. The other possibility that C_N could not have been committed at all directly contradicts the fact that C_N is committed. Therefore, our supposition that L' is less than C_N must be false.

Once a node has recovered its LLE, it can participate in elections. If an already recovered node or a node that has not failed so far becomes the leader (for example, S_1 or S_5 in Figure 4.4(a)(i)), it will already have the LLE-MAP, which it can use in subsequent replication requests. On the other hand, if a recently recovered node becomes the leader (for example, S_3 in Figure 4.4(a)(i)), then it needs to construct the LLE-MAP values for other nodes. To enable this construction, during an election, the voting nodes send their LLE-MAP to the candidate as part of the vote responses. Using these responses, the candidate constructs the LLE-MAP value for each node by picking the maximum LLE of that node from a bare-minority responses.

Data recovery. Once a node has successfully recovered its LLE, it needs to recover the actual data. If the recovering node becomes the follower, it simply fetches the latest data from the leader. In contrast, if the recovering node becomes the leader, it recovers the data up to the recovered LLE from the followers.

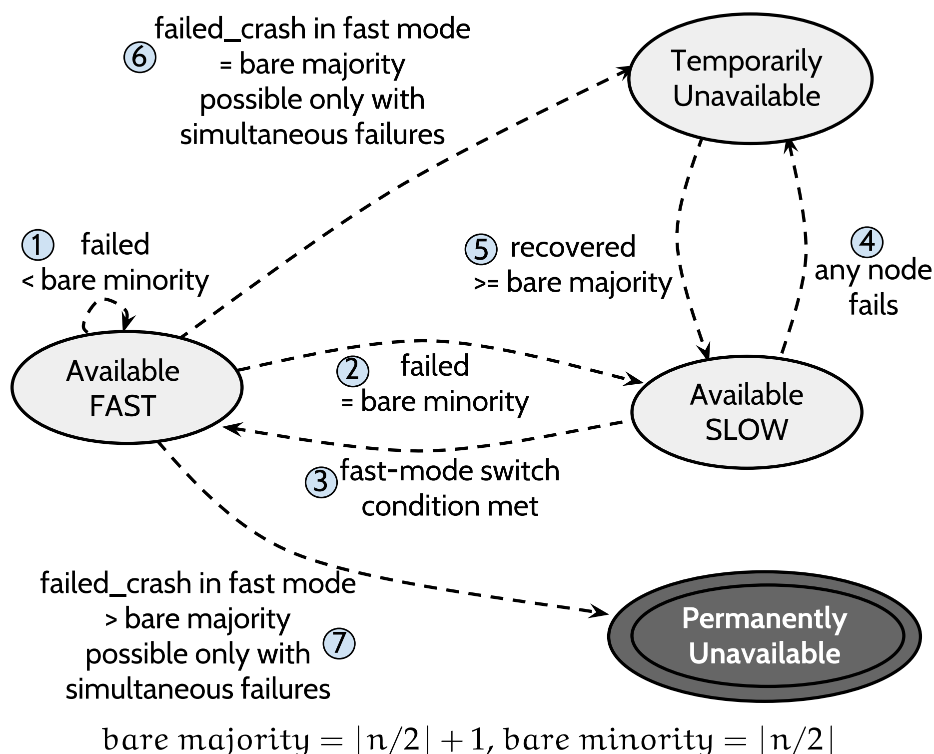


Figure 4.5: **Saucr Summary and Guarantees.** *The figure summarizes how Saucr works under failures and the guarantees it provides.*

4.2.7 Summary and Guarantees

We use Figure 4.5 to summarize how Saucr works and the guarantees it offers; a node fails either by crashing or by becoming unreachable over the network. We guide the reader through the description by following the sequence numbers shown in the figure. ① At first, we assume all nodes are in recovered state; in this state, Saucr operates in the fast mode; when nodes fail, Saucr stays in the fast mode as long as the number of nodes failed is less than a bare minority. ② After a bare minority of nodes fail, Saucr switches to slow mode. ③ Once in slow mode, if one or more nodes recover and respond promptly for a few requests, Saucr transitions back to fast mode. ④ In slow mode, if any node fails, Saucr becomes temporarily

unavailable. ⑤ Once a majority of nodes recover, the system becomes available again.

To explain further transitions, we differentiate non-simultaneous and simultaneous crashes and network partitions. In the presence of non-simultaneous crashes, nodes will have enough time to detect failures; the leader can detect follower crashes and switch to slow mode and followers can detect the leader's crash and flush to disk. Thus, despite any number of non-simultaneous crashes, Saucr always transitions through slow mode. Once in slow mode, the system provides strong guarantees.

However, in the presence of simultaneous crashes, many nodes could crash instantaneously while in fast mode; in such a scenario, Saucr cannot always transition through slow mode. ⑥ If the number of nodes that crash in fast mode does not exceed a majority, Saucr will only be temporarily unavailable; in this case, at least a bare minority will be in recovered state or will have previously crashed in slow mode making crash recovery possible (as described in §4.2.6). ⑦ In rare cases, more than a bare majority of nodes may crash in fast mode, in which case, crash recovery is not possible: the number of nodes that are in recovered state or previously crashed in slow mode will be less than a bare minority. During such simultaneous crashes, which we believe are extremely rare, Saucr remains unavailable.

In the presence of partitions, all nodes could be alive, but partitioned into two; in such a case, the minority partition would be temporarily unavailable while the other partition will safely move to slow mode if a bare majority are connected within the partition. The nodes in the minority partition would realize they are not connected to the leader and flush to disk. Both of these actions guarantee durability and prevent future unavailability.

4.3 Implementation

We have implemented situation-aware updates and crash recovery in Apache ZooKeeper (v3.4.8). We now describe the most important implementation details.

Storage layer. ZooKeeper maintains an on-disk log to which the updates are appended. ZooKeeper also maintains snapshots and meta information (e.g., current epoch). We modified the log-update protocol to not issue `fsync` calls synchronously in fast mode. Snapshots are periodically written to disk; because the snapshots are taken in the background, foreground performance is unaffected. The meta information is always synchronously updated. Fortunately, such synchronous updates happen rarely (only when the leader changes), and thus do not affect common-case performance. In addition to the above structures, SAUCR maintains the *fast-switch-entry* in a separate file and synchronously updates it the first time when the node processes an entry in the fast mode. In slow mode, the `LLE-MAP` is synchronously persisted. SAUCR maintains the map at the head of the log file. The *latest-on-disk-entry* for a node is its own entry in the persistent `LLE-MAP` (`LLE-MAP` is keyed by node-id).

Replication. We modified the *QuorumPacket* [25] (which is used by the leader for replication) to include the mode flag and the `LLE-MAP`. The leader transitions to fast mode after receiving three consecutive successful replication acknowledgements from more than a bare majority.

Failure Reaction. In our implementation, the nodes detect failures through missing heartbeats, missing responses, and broken socket connections. Although quickly reacting to failures and flushing or switching modes is necessary to prevent data loss or unavailability, hastily declaring a node as failed might lead to instability. For example, if a follower runs for an election after missing just one heartbeat from the leader, the system may often change leader, affecting progress. SAUCR's implementation avoids this scenario as follows. On missing the first heartbeat from the leader,

the followers *suspect* that the leader might have failed and so quickly react to the suspected failure by flushing their buffers. However, they conservatively wait for a handful of missing heartbeats before *declaring* the leader as failed and running for an election. Similarly, while the leader initiates a mode switch on missing the first heartbeat response, it waits for a few missing responses before declaring the follower as failed. If a majority of followers have not responded to a few heartbeats, the leader steps down and becomes a candidate.

Recovery Protocol. We modified the leader election protocol so that a node recovering from a fast-mode crash first recovers its LLE before it can participate in elections. A responding node correctly handles LLE-query from a node and replication requests from the leader that arrive concurrently. If a node that recovers from a fast-mode crash becomes the leader, it fetches the data items up to its LLE from others. However, due to the background flushes, several items might already be present on the disk; the node recovers only the missing items. The responding node batches several items in its response.

4.4 Evaluation

We now evaluate the durability, availability, and performance of our SAUCR implementation.

4.4.1 Durability and Availability

To evaluate the guarantees of SAUCR, we developed a cluster crash-testing framework. The framework first generates a graph of all possible cluster states as shown in Figure 4.6. Then, it generates a set of *cluster-state sequences*. For instance, 12345 → 345 → 45 → 1245 → 1 → 13 → 12345 is one such sequence. In this sequence, at first, all five nodes are alive; then, two nodes (1 and 2) crash; then, 3 crashes; next, 1 and 2 recover; then 2,

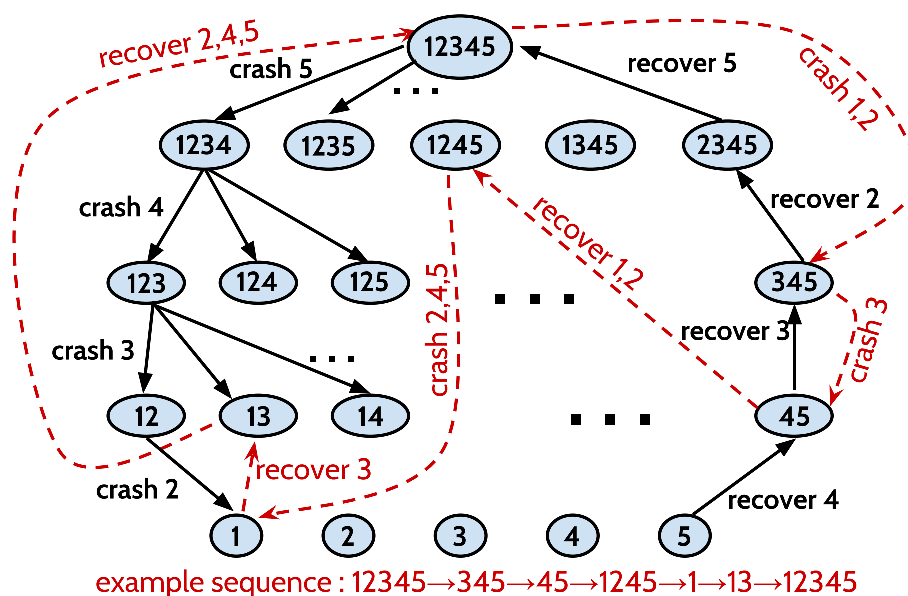


Figure 4.6: **Cluster-State Sequences.** *The figure shows the possible cluster states for a five-node cluster and how cluster-state sequences are generated. One example cluster-state sequence is traced.*

4, 5 crash; 3 recovers; finally, 2, 4, 5 recover. To generate a sequence, we start from the root state where all nodes are alive. We visit a child with a probability that decreases with the length of the path constructed so far, and the difference in the number of alive nodes between the parent and the child. We produced 1264 such sequences (498 and 766 for a 5-node and 7-node cluster, respectively).

The cluster-state sequences help test multiple update and recovery code paths in Saucr. For example, in the above sequence, 12345 would first operate in fast mode; then 345 would operate in slow mode; then 1245 would operate in fast mode; 1 would flush to disk on detecting that other nodes have crashed; in the penultimate state, 3 would recover from a slow-mode crash; in the last state, 2, 4, and 5 would recover from a fast-mode crash.

Within each sequence, at each intermediate cluster state, we insert

new items if possible (if a majority of nodes do not exist, we cannot insert items). $12345^a \rightarrow 345^b \rightarrow 45 \rightarrow 1245^c \rightarrow 1 \rightarrow 13 \rightarrow 12345^d$ shows how entries a-d are inserted at various stages. In the end, the framework reads all the acknowledged items. If the cluster does not become available and respond to the queries, we flag the sequence as unavailable for the system under test. If the system silently loses the committed items, then we flag the sequence as data loss.

We subject the following four systems to the cluster-crash sequences: memory-durable ZK (ZooKeeper with the *forceSync* flag turned off), VR (viewstamped replication), disk-durable ZK (ZooKeeper with *forceSync* turned on), and finally SAUCR. Existing VR implementations [230] do not support a read/write interface, preventing us from directly applying our crash-testing framework to them. Therefore, we developed an *ideal* model of VR that resembles a perfect implementation.

Non-simultaneous Crashes

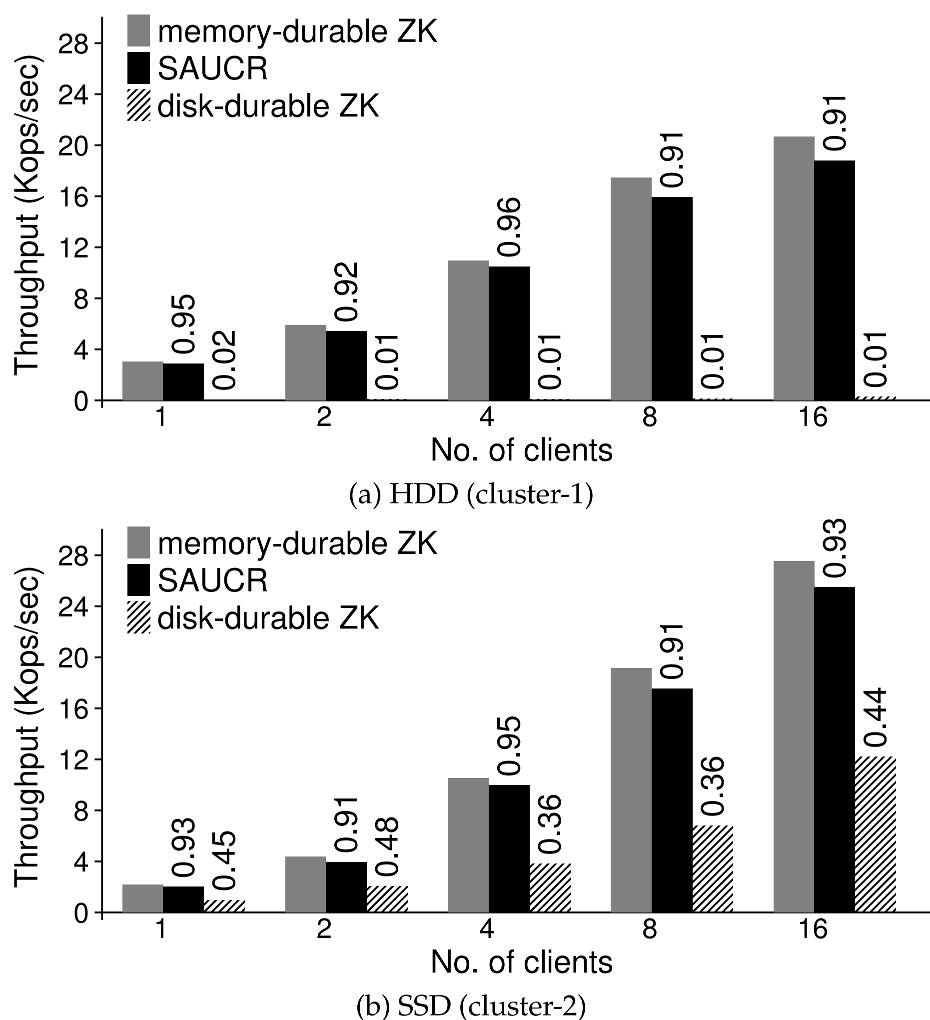
We first test all sequences considering that failures are non-simultaneous. For example, when the cluster transitions from 12345 to 345, we crash nodes 1 and 2 one after the other (with a gap of 50 ms). Table 4.2 shows the results. As shown, the memory-durable ZK loses data in about 50% and 40% of the cases in the 5-node and 7-node tests, respectively. The ideal VR model does not lose data; however, it leads to unavailability in about 90% and 75% of the cases in the 5-node and 7-node tests, respectively. As expected, disk-durable ZooKeeper is safe. In contrast to memory-durable ZK and VR, SAUCR remains durable and available in all cases. Because failures are non-simultaneous in this test, the leader detects failures and switches to slow mode; similarly, the followers quickly flush to disk if the leader crashes, leading to correct behavior.

System	Nodes	Non-simultaneous Total	Correct	Unavailable	Data loss	Simultaneous Scenario	Total	Correct	Unavailable	Data loss
ZK-mem	5	498	248	0	250	n/a	498	248	0	250
	7	766	455	0	311	n/a	766	455	0	311
VR-ideal	5	498	28	470	0	n/a	498	28	470	0
	7	766	189	577	0	n/a	766	189	577	0
ZK-disk	5	498	498	0	0	n/a	498	498	0	0
	7	766	766	0	0	n/a	766	766	0	0
SAUCR	5	498	498	0	0	other	475	475	0	0
						!min-rec	23	0	23	0
	7	766	766	0	0	other	725	725	0	0
						!min-rec	41	0	41	0

Table 4.2: **Durability and Availability.** *The table shows the durability and availability of memory-durable ZK (ZK-mem), VR (VR-ideal), disk-durable ZK (ZK-disk), and* SAUCR*. !min-rec denotes that only less than a bare minority are in recovered state.*

Simultaneous Crashes

We next assume that failures are simultaneous. For example, if the cluster state transitions from 124567 to 12, we crash all four nodes at the same time, without any gap. Note that during such a failure, SAUCR would be operating in fast mode and suddenly many nodes would crash simultaneously, leaving behind less than a bare minority. In such cases, less than a bare minority would be in the *recovered* state; SAUCR cannot handle such cases. Table 4.2 shows the results. As shown, memory-durable ZK loses data in all cases in which it lost data in the non-simultaneous test. This



(a) HDD (cluster-1)

(b) SSD (cluster-2)

Figure 4.7: **Micro-benchmarks.** *(a) and (b) show the update throughput on memory-durable ZK, SAUCR, and disk-durable ZK on HDDs and SSDs, respectively. Each request is 1KB in size. The number on top of each bar shows the performance normalized to that of memory-durable ZK.*

is because memory-durable ZK loses data, irrespective of the simultaneity of the crashes. Similarly, VR is unavailable in all the cases where it was unavailable in the non-simultaneous crash tests. As expected, disk-durable ZK remains durable and available. SAUCR remains unavailable in

a few cases by its design.

4.4.2 Performance

We conducted our performance experiments on two clusters (cluster-1: HDD, cluster-2: SSD), each with five machines. The HDD cluster has a 10-Gbps network, and each node is a 20-core Intel Xeon CPU E5-2660 machine with 256 GB memory running Linux 4.4, with a 1-TB Seagate ST1200MM0088 HDD. The SSD cluster has 10-Gbps network, and each node is a 20-core Intel E5-2660 machine with 160 GB memory running Linux 4.4, with a 480-GB Intel SSDSC2BB480G7K SSD. Numbers reported are the average over five runs.

Update Micro-benchmark

We now compare SAUCR's performance against memory-durable ZK and disk-durable ZK. We conduct this experiment for an update-only micro-benchmark.

Figure 4.7(a) and (b) show the results on HDDs and SSDs, respectively. As shown in the figure, SAUCR's performance is close to the performance of memory-durable ZK (overheads are within 9% in the worst case). Note that SAUCR's performance is close to memory-durable ZK but not equal; this small gap exists because, in the fast mode, SAUCR commits a request only after four nodes (majority + 1) acknowledge, while memory-durable ZK commits a request after three nodes (a bare majority) acknowledge. Although the requests are sent to the followers in parallel, waiting for acknowledgment from one additional follower adds some delay. Compared to disk-durable ZK, as expected, both memory-durable ZK and SAUCR are significantly faster. On HDDs, they are about 100× faster. On SSDs, however, the performance gap is less pronounced. For instance, with a single client, memory-durable ZK and SAUCR are only about 2.1× faster than

disk-durable ZK. We found that this inefficiency arises because of software overheads in ZooKeeper's implementation that become dominant atop SSDs.

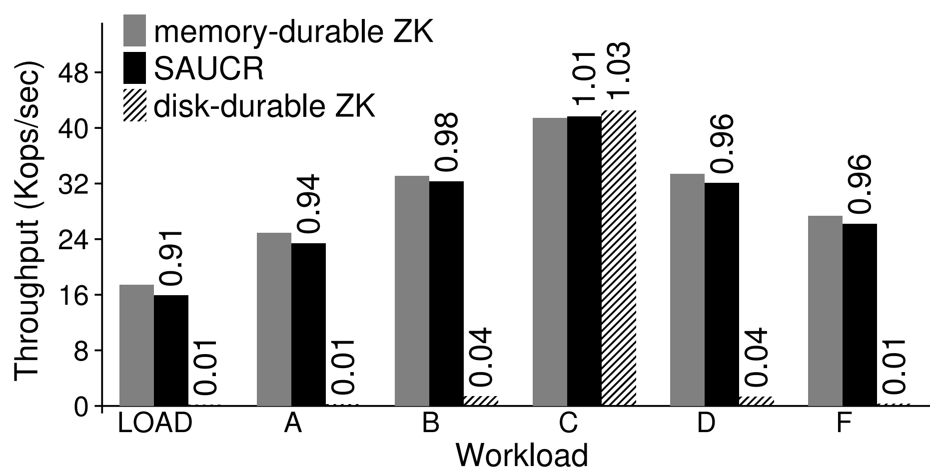
YCSB Workloads

We now compare the performance of Saucr against memory-durable ZK and disk-durable ZK across the following six YCSB [63] workloads: load (all writes), A (w:50%, r:50%), B (w:5%, r:95%), C (only reads), D (read latest, w:5%, r:95%), F (read-modify-write, w:50%, r:50%). We use 1KB requests.

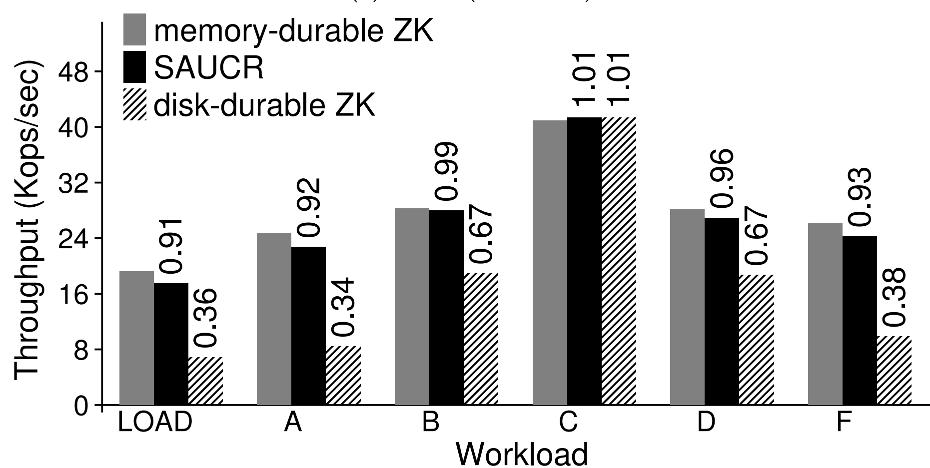
Figure 4.8(a) and (b) show the results on HDDs and SSDs, respectively. For all workloads, Saucr closely matches the performance of memory-durable ZK; again, the small overheads are a result of writing to one additional node. For write-heavy workloads (load, A, F), Saucr's performance overheads are within 4% to 9% of memory-durable ZK. For such workloads, memory-durable ZK and Saucr perform notably better than disk-durable ZK (about 100× and 2.5× faster on HDDs and SSDs, respectively). For workloads that perform mostly reads (B and D), Saucr's overheads are within 1% to 4% of memory-durable ZK. For such read-heavy workloads, memory-durable ZK and Saucr are about 25× and 40% faster than disk-durable ZK on HDDs and SSDs, respectively. For the read-only workload (C), all three systems perform the same on both HDDs and SSDs because reads are served only from memory.

Performance Under Failures

In all our previous performance experiments, we showed how Saucr performs in its fast mode (without failures). When failures arise and if only a bare majority of nodes are alive, Saucr switches to the slow mode until enough nodes recover. Figure 4.9 depicts how Saucr detects failures



(a) HDD (cluster-1)



(b) SSD (cluster-2)

Figure 4.8: **Macro-benchmarks.** *The figures show the throughput under various YCSB workloads for memory-durable ZK,* Saucr*, and disk-durable ZK for eight clients. The number on top of each bar shows the performance normalized to that of memory-durable ZK.*

and switches to slow mode when failures arise. However, when enough nodes recover from the failure, Saucr switches back to fast mode.

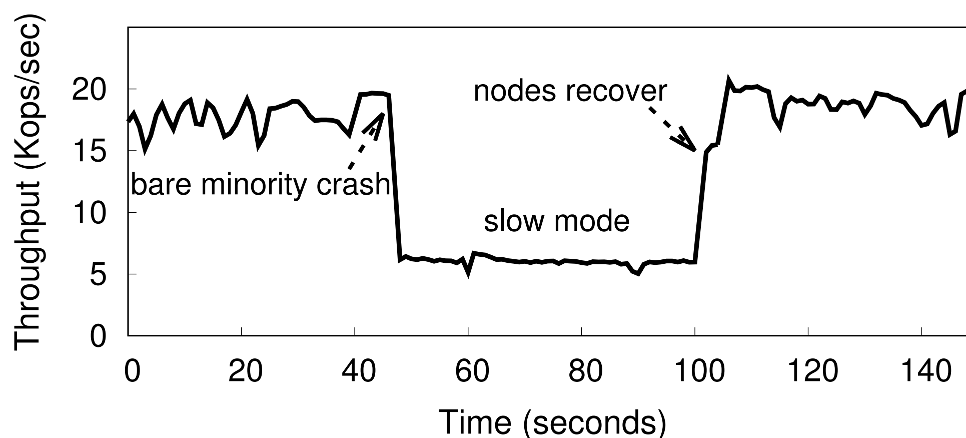


Figure 4.9: **Performance Under Failures.** *The figure shows* Saucr*'s performance under failures; we conduct this experiment with eight clients running an update-only workload on SSDs.*

4.4.3 Heartbeat Interval vs. Performance

Saucr uses heartbeats to detect failures. We now examine how varying the heartbeat interval affects workload performance. Intuitively, short and aggressive intervals would enable quick detection but lead to worse performance. Short intervals may degrade performance for two reasons: first, the system would load the network with more packets; second, the Saucr nodes would consider a node as failed upon a missing heartbeat-/response when the node was merely slow and thus react spuriously by flushing to disk or switching to slow mode.

To tackle the first problem, when replication requests are flowing actively, Saucr treats the requests themselves as heartbeats; further, we noticed that even when the heartbeat interval is lower than a typical replication-request latency, the additional packets do not affect the workload performance significantly. The second problem of spurious reactions can affect performance.

For the purpose of this experiment, we vary the heartbeat interval

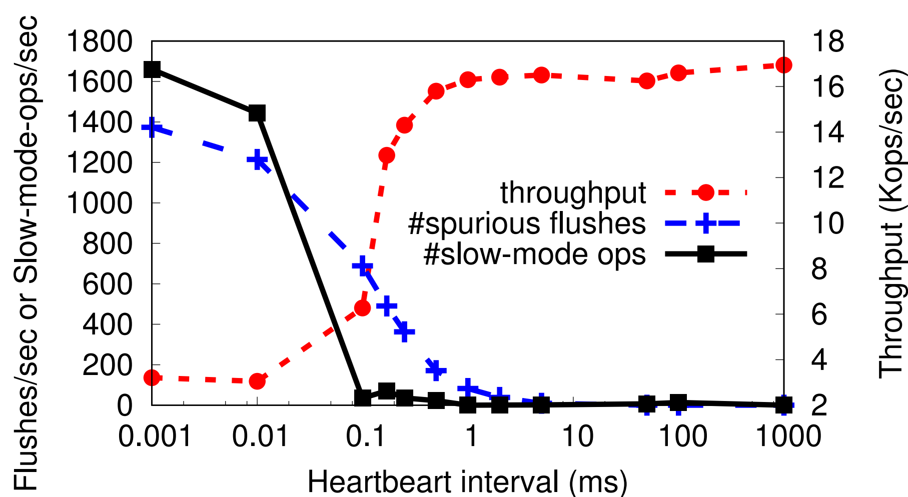


Figure 4.10: **Heartbeat Interval vs. Performance.** *The figure shows how varying the heartbeat interval affects performance. The left y-axis shows the average number of flushes issued by a follower per second or the average number of requests committed in slow mode by the leader per second. We measure the performance (right y-axis) by varying the heartbeat interval (x-axis). We conduct this experiment with eight clients running the YCSB-load workload on SSDs.*

from a small (and unrealistic) value such as 1 µs to a large value of 1 second. We measure three metrics: throughput, the number of requests committed in slow mode (caused by the leader suspecting follower failures due to a missing heartbeat response), and the number of flushes issued by a follower (caused by followers suspecting a leader failure due to a missing heartbeat). Figure 4.10 shows the result. As shown, when the interval is equal to or greater than 1 ms, the workload performance remains mostly unaffected. As expected, with such reasonably large intervals, even if the nodes are slow occasionally, the likelihood that a node will not receive a heartbeat or a response is low; thus, the nodes do not react spuriously most of the times. As a result, only a few spurious flushes are issued by the followers, and very few requests are committed in slow mode. In contrast, when the interval is less than 1 ms, the SAUCR nodes react more aggressively, flushing more often and committing many re-

quests in slow mode, affecting performance. In summary, for realistic intervals of a few tens of milliseconds (used in other systems [81]) or even for intervals as low as 1 ms, workload performance remains unaffected.

Finally, although the nodes react aggressively (with short intervals), they do not declare a node as failed because there are no actual failures in this experiment. As a result, we observe that the leader does not step down and the followers do not run for an election.

4.4.4 Correlated Failure Reaction

We now test how quickly Saucr detects and reacts to a correlated failure that crashes all the nodes. On such a failure, if at least a bare minority of nodes flush the data to disks before all nodes crash, Saucr will be able to provide availability and durability when the nodes later recover. For this experiment, we use a heartbeat interval value of 50 ms. We conduct this experiment on a five-node cluster in two ways.

First, we crash the active leader and then successively crash all the followers. We vary the time between the individual failures and observe how many followers detect and flush to disk before all nodes crash. For each failure-gap time, we run the experiment five times and report the average number of nodes that safely flush to disk. Figure 4.11 shows the result: if the time between the failures is greater than 30 ms, then at least a bare minority of followers always successfully flush the data, ensuring availability and durability.

Second, we crash the followers, one after the other. In this case, the leader detects the failures and switches to slow mode. As shown in the figure, if the time between the individual failures is greater than 50 ms, the system will be available and durable after recovery. As we discussed earlier (§4.1.3), in a real deployment, the time between individual failures is almost always greater than 50 ms; therefore, in such cases, with a heartbeat interval of 50 ms, Saucr will always remain safe.

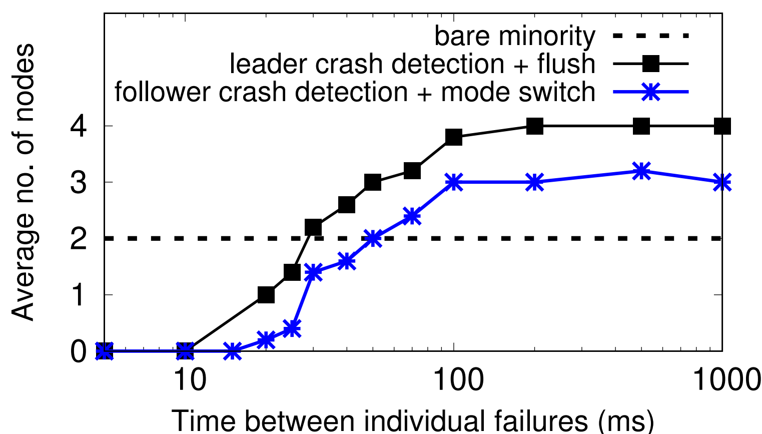


Figure 4.11: **Correlated Failure Reaction.** *The figure shows how quickly* Saucr *reacts to correlated failures; the y-axis denotes the number of nodes that detect and flush to disk before all nodes crash when we vary the time between the individual failures (x-axis). We conduct this experiment on SSDs.*

Note that we run this experiment with a 50-ms heartbeat interval; shorter intervals (such as 1 ms used in Figure 4.10) will enable the system to remain durable and available (i.e., a bare minority or more nodes would safely flush or switch to slow mode) even when the failures are only a few milliseconds apart.

4.5 Discussion

We now discuss two concerns related to Saucr's adoption in practice. First, we examine whether Saucr will offer benefits with the advent of faster storage devices such as non-volatile memory (NVM). Second, we discuss whether applications will be tolerant of having low throughput when Saucr operates in slow mode.

Faster Storage Devices. The reliability of memory-durable approaches can be significantly improved if every update is forced to disk. However, on HDDs or SSDs, the overhead of such synchronous persistence is pro-

hibitively expensive. New storage devices such as NVMe-SSDs and NVM have the potential to reduce the cost of persistence and thus improve reliability with low overheads. However, even with the advent of such faster storage, we believe SAUCR has benefits for two reasons.

First, although NVMe-SSDs are faster than HDDs and SSDs, they are not as fast as DRAM. For example, a write takes 30 µs on Micron NVMe-SSDs which is two orders of magnitude slower than DRAM [57] and thus SAUCR will have performance benefits compared to NVMe-SSDs. While NVM and DRAM exhibit the same latencies for reads, NVM writes are more expensive (roughly by a factor of 5) [122, 234]. Further, writing a few tens of kilobytes (as a storage system would) will be slower than published numbers that mostly deal with writing cachelines. Hence, even with NVMs, SAUCR will demonstrate benefit.

Second, and more importantly, given the ubiquity of DRAM and their lower latencies, many current systems and deployments choose to run memory-only clusters for performance [50, 126], and we believe this trend is likely to continue. SAUCR would increase the durability and availability of such non-durable deployments significantly without affecting their performance at no additional cost (i.e., upgrading to new hardware).

Low Performance in Slow Mode. Another practical concern regarding SAUCR's use in real deployments is that of the low performance that applications may experience in slow mode. While SAUCR provides low performance in slow mode, we note that this trade-off is a significant improvement over other existing methods that can either lead to permanent unavailability or lose data. Further, in a shared-storage setting, we believe many applications with varying performance demands will coexist. While requests from a few latency-sensitive applications may time out, SAUCR allows other applications to make progress without any hindrance. Furthermore, in slow mode, only update requests pay the performance penalty, while most read operations can be served without any

overheads (i.e., at about the same latency as in the fast mode). Finally, this problem can be alleviated with a slightly modified system that can be reconfigured to include standby nodes when in slow mode for a prolonged time. Such reconfiguration would enable the system to transition out of the slow mode quickly.

4.6 Summary and Conclusions

Fault-tolerant replication protocols are the foundation upon which many data-center systems and applications are built. Such a foundation needs to perform well, yet also provide a high level of reliability. Existing replication approaches statically fix how updates will be committed: they always update and recover in a constant way, regardless of the situation. This situation-obliviousness leads to poor performance or reliability.

In this chapter, we presented situation-aware updates and crash recovery (Saucr), a new approach to replication within a distributed system. Saucr reacts to failures and adapts to current conditions, improving durability and availability while maintaining high performance.

We implemented a prototype of Saucr in ZooKeeper. Through a series of robustness test, we showed that Saucr provides improved durability and availability compared to memory-durable approaches. Saucr's reliability improvements come at low cost: Saucr's overheads are within 0%-9% of memory-durable ZooKeeper across six different YCSB workloads. Compared to the disk-durable ZooKeeper, with a slight reduction in availability in rare cases, SAUCR improves performance by 25× to 100× on HDDs and 2.5× on SSDs. We believe such a situation-aware distributed update and recovery protocol can serve as a better foundation upon which reliable and performant systems can be built.

We conclude by making two broad remarks about the work presented in this chapter. First, our design of Saucr emphasizes the importance of

paying careful attention to how actual failures occur and using those insights to design systems. This way of thinking about systems design can result in solutions that offer properties that seem to be at odds with each other. In SAUCR, our attention to how correlated failures arise in data centers yielded a dynamic replication scheme that provides both high performance and strong reliability to crashes. Next, our design of SAUCR shows that hybrid approaches, which we believe is an effective systems-design technique in general, is a good choice for distributed updates and recovery too. We believe it may be worthwhile to look at other important protocols and systems where such hybrid approaches may be suitable.

5 File-system Crash Behaviors in Distributed Systems

In this chapter, we analyze the effects of file-system crash behaviors on modern distributed storage systems. When a node recovers from a crash, the common expectation is that the data stored by the node would be recoverable. Unfortunately, as we discussed (§2.3.3), the local file system (which the node uses to store user data) complicates this situation. We examine if and how such file-system crash behaviors affect distributed storage systems. We perform our study under a special type of correlated failure scenario in which all replicas of a particular data shard crash together and recover at a later point.

To reason about the persistent states that can arise on the nodes during such a failure, we develop Pace. Pace models local file systems at individual replicas using an *abstract persistence model* (APM). Pace uses *protocol-specific knowledge* to reduce the exploration state space by systematically choosing a subset of nodes to introduce file-system crash behaviors modeled by the APM. We applied Pace to eight distributed storage systems and discovered 26 new vulnerabilities that have severe consequences such as data loss and unavailability. This chapter is based on the paper, *Correlated Crash Vulnerabilities*, published in OSDI 16 [12].

We first discuss the failure model that we consider and explain the global persistent states across nodes that we intend to capture (§5.1). We

then describe the design of Pace and show how it uses a set of generic rules to prune the state space (§5.2). Next, we present a detailed study of the vulnerabilities that we found using Pace (§4.3). We then discuss how the discovered vulnerabilities can be potentially fixed (§5.4). Finally, we summarize and conclude (§5.5).

5.1 Studying the Effects of File-system Crash Behaviors

In this section, we first describe the failure model that we consider and build arguments for why the considered failure model is important. Next, we explain the system states we explore to find if a distributed storage system has vulnerabilities.

5.1.1 Failure Model

Components in a distributed system can fail in various ways [98]. Most practical systems do not handle Byzantine failures [131] where individual components may give conflicting information to different parts of the system. However, they handle fail-recover failures [98] where components can crash at any point in time and recover at any later point in time after the cause of the failure has been repaired. When a node crashes and recovers, all its in-memory state is lost; the node is left only with its persistent state. Our study considers only such fail-recover failures.

File-system Crash Behaviors

When a node recovers from a crash, it can encounter many possible persistent states depending upon the file system used by the node. We explain such persistent states that can arise at a node using Figure 5.1. The figure shows the file-system operations on a single node P (which is a part

```
Node P

# State Pϕ
write(fd, “foo”, 3)
# State P1
write(fd, “baz”, 3)
# State P2
```

Figure 5.1: **Persistent States in a Single Node.** *The figure shows file-system operations and the persistent states on a single node in a distributed system.*

of a distributed system). As shown, the node performs two write operations to a file. For now, assume the file system at P is synchronous (i.e., operations are persisted in order and immediately to the disk). If a crash occurs, there are only three possible persistent states that can arise when the node recovers from the crash: P_ϕ (the node crashed before the first write), P_1 (the node crashed after the first write but before the second), or P_2 (the node crashed after the second write).

In reality, however, most modern file systems buffer the writes issued by the node. Depending on which exact file system and mount options are in use, the file system may reorder some (or many) updates [28, 181]. With this asynchrony and reordering introduced by the file system, it is possible for the second write `baz` to reach the disk before the first write `foo`. Thus, when P recovers from the crash, it may find itself in a persistent state where the effects of second write are present but not the first.

The reordering of writes by the file system, as explained above, is well understood by experienced developers. To avoid such reordering, developers force writes to disk by carefully issuing `fsync` on a file as part of the update protocol. Although some common behaviors such as reordering of writes are well understood, there are subtle behaviors that application developers find hard to reason about. For example, the following subtle behavior is not well documented: if a crash happens when appending a single block of data to a file in ext4 writeback mode, the file may contain

garbage on reboot. Thus, in our example, upon such a file system, if the first write spans a block and if P crashes during the write, then it may see that the file is filled with garbage upon recovery.

Recent research has discovered that such behaviors of file systems affect the correctness of single-machine applications, causing them to corrupt or lose user data [9, 181]. Distributed storage systems also face the same problem as each replica uses its local file system to store user data, and untimely crashes may leave the node in an inconsistent state. However, distributed systems have more opportunities for recovery as redundant copies of data exist on other nodes.

Correlated Crashes

Note that our previous discussion about Figure 5.1 explains about the states that are possible on a single node in the distributed system. This is how one individual node crashes and recovers; other nodes may continue to function and make progress. However, we focus on a much restricted failure scenario in this study. Specifically, we aim to analyze the effects of file-system crash behaviors under a special correlated crash scenario where *all* replicas of a particular shard of data fail together and none of the replicas react to the failure as they are occurring. During such a failure, the local file systems may produce unexpected states at one or more nodes in the distributed system.

Our failure model is not intended to reason about scenarios where only a subset of replicas of a particular data shard crash and recover by themselves. Also, the vulnerabilities we find with our correlated failure model do not apply to a geo-replicated setting; in such a setting, conscious decisions place replicas such that one power failure cannot affect all replicas at the same time. While correlated failures are less problematic in such settings, the storage systems we examine in this study are heavily tested, and the common expectation is that these systems should

be reliable irrespective of how they are deployed and the probability of failures. Further, many deployments are not geo-replicated and thus may expect strong guarantees even in the presence of correlated crashes. Overall, crash-correctness should be deeply ingrained in these systems regardless of deployment decisions.

5.1.2 Distributed Crash States

Now we explain the global system states that result due to correlated crashes and file-system crash behaviors. As we explained, after a crash and subsequent reboot, a node is left only with its persistent data. The focus of our study is in checking only the resulting persistent states when failures happen. The global states that we capture are similar to distributed snapshots [51] described by Chandy and Lamport. The main difference between a generic distributed snapshot and a global persistent state is that the latter consists only of the on-disk state and not the in-memory state of the machines. Moreover, since network channels do not affect persistent on-disk state, our global persistent states do not keep track of them.

To understand the persistent states that we capture, consider a cluster of three machines named A, B, and C. Assume that the initial persistent states of these machines are A_ϕ, B_ϕ, and C_ϕ, respectively. Assume that a workload W run on this cluster transitions the persistent states to A_f, B_f, and C_f, respectively. For instance, W could be a simple workload that inserts a new key-value pair into a replicated key-value store running on A, B, and C. Notice that the persistent state of all nodes goes through a transition before arriving at the final states A_f, B_f, and C_f. A correlated crash may happen at any time while W runs, and after a reboot, the persistent state of a node X may be any intermediate state between X_ϕ and X_f where X can be A, B, or C. For simplicity, we refer to this collection of persistent states across all nodes as global persistent state or simply global state. If

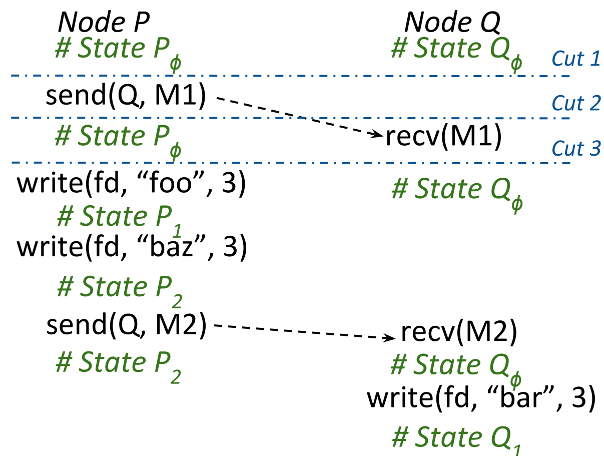


Figure 5.2: **A Simple Distributed Protocol.** *The figure shows a simple distributed protocol. Annotations show the persistent state after performing each operation. Dash dot lines show different cuts.*

a particular global state G can occur in an execution, we call G a *reachable* global state.

The reachability of a global state depends on two factors: the order in which messages are exchanged between nodes and the local file systems of the nodes. To illustrate the first factor, consider a distributed protocol shown in Figure 5.2. In this protocol, node P starts by sending message `M1`, then writes `foo` and `baz` to a file, and then sends another message `M2` to node Q. Node Q receives `M1` and `M2` and then writes `bar` to a file. Assume that the file system at P and Q is synchronous.

Assume that the initial persistent state of P was P_ϕ and Q was Q_ϕ. After performing the first and second write, P moves to P_1 and P_2, respectively. Similarly, Q moves to Q_1 after performing the write. Notice that $< P_\phi, Q_\phi >$ is a reachable global persistent state as P could have crashed before writing to the file and Q could have crashed before or after receiving the first message. Similarly, $< P_2, Q_1 >$ and $< P_2, Q_\phi >$ are globally reachable persistent states.

In contrast, $< P_\phi, Q_1 >$ and $< P_1, Q_1 >$ are unreachable persistent

states as it is not possible for Q to have updated the file without P sending the message to it. Intuitively, global states that are not reachable in an execution are logically equivalent to inconsistent cuts in a distributed system [30]. For example, $< P_\phi, Q_1 >$ and $< P_1, Q_1 >$ are inconsistent cuts because the recv of M2 is included in the cut but the corresponding send is excluded from the cut. Also, network operations such as send and recv do not affect the persistent state. For example, the three different cuts shown in Figure 5.2 map onto the same persistent state $< P_\phi, Q_\phi >$.

Next, we consider the fact that the local file systems at P and Q also influence the global states. For example, with the reordering introduced by the file system at P, it is possible for the second write baz to reach the disk before the first write foo. Also, it is possible for P to crash after baz is persisted and the message is sent to Q, but before foo reaches the disk. In such a state of P, it is possible for Q to have either reached its final state Q_1 or crash before persisting bar and so remain in Q_ϕ. All these states are globally reachable.

5.2 Protocol-Aware Crash Explorer

To examine if distributed storage systems violate user-level guarantees, we build a generic crash exploration framework, Pace. Pace systematically generates persistent states that can occur in a distributed execution in the presence of correlated crashes. Pace then uses an abstract persistence model to introduce file-system crash behaviors on one or more nodes. Some vulnerabilities that we discover are exposed only if a particular file-system operation is reordered on all replicas while some vulnerabilities are exposed even when the reordering happens on a single replica. Using observations from how vulnerabilities are exposed and a little knowledge about the distributed protocol, we make our exploration *protocol-aware*. Using this awareness, Pace can prune the search space while finding as

many vulnerabilities as a brute-force search. To explain how protocol-aware exploration works, we first describe the design of our crash exploration framework.

5.2.1 Design and Implementation Overview

Pace is easy to use and can be readily applied to any distributed storage system. Pace needs a workload script and a checker script as inputs. An example workload and checker are shown in Listing 5.1. For many modern distributed systems, a group of processes listening on different ports can act as a cluster. For systems that do not allow this convenience, we use a group of Docker [75] containers on the same machine to serve as the cluster. In either case, Pace can test the entire system on a single machine. Pace is implemented in around 5500 lines of code in Python.

Figure 5.3 shows a typical workflow of Pace. To begin, Pace starts the cluster with system call tracing, runs the workload, and then stops the cluster after the workload is completed. Pace parses the traces obtained and identifies cross node dependencies such as a `send` on one node and the corresponding `recv` on some other node. After the traces are parsed and cross node dependencies established, Pace replays the trace to generate different persistent crash states that can occur in the traced execution. A system-specific checker script is run on top of each crash state; the checker script asserts whether user-level guarantees (e.g., committed data should not be corrupted or lost) hold. Any violations in such assertions are reported as vulnerabilities. We next discuss what correlated crash states can occur in a distributed execution and how we generate them.

5.2.2 Crash States

We use a running example of a ZooKeeper cluster executing an update workload for further discussion. Pace produces a diagrammatic repre-

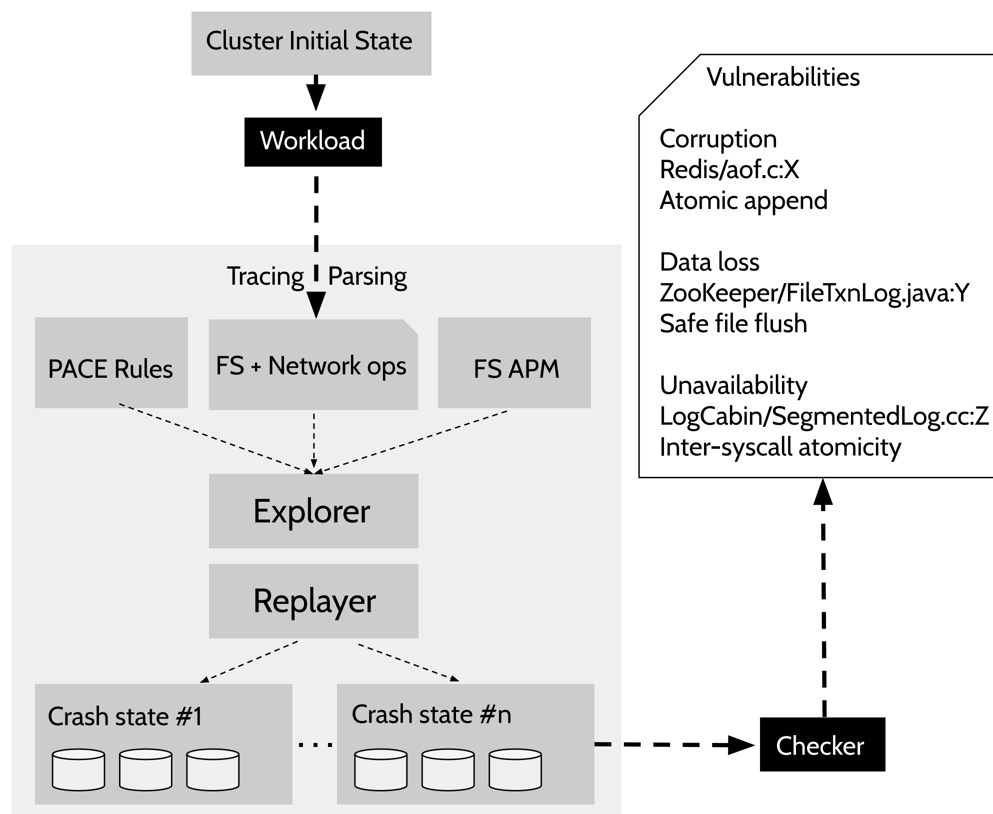


Figure 5.3: **PACE Workflow.** *The figure shows* PACE*'s workflow. First,* PACE *traces a workload on an initial cluster state, capturing file-system and network operations.* PACE *imposes the file system behavior through the FS APMs. Then, it uses its exploration rules to produce many distributed crash states. Finally, each distributed crash state is verified by a checker which restarts the cluster from the crash state and performs various checks (e.g., are committed data items available?). If the checker finds a violation, it reports them as vulnerabilities, pointing to the source-code lines responsible for the vulnerability.*

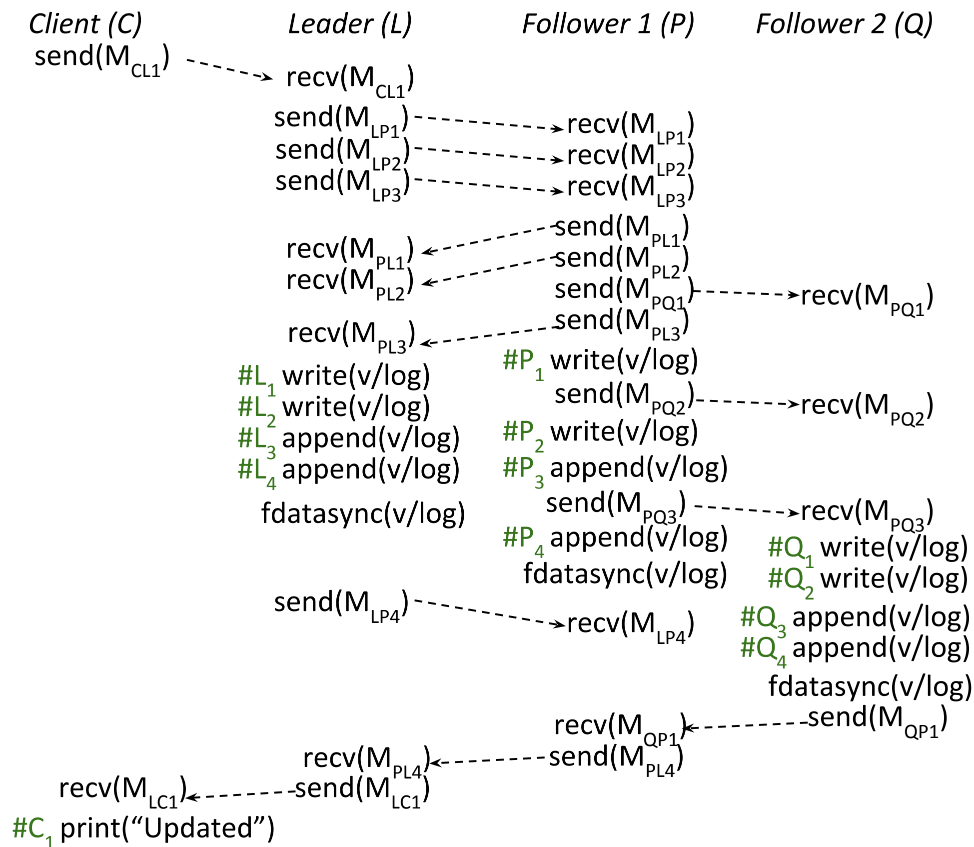


Figure 5.4: **ZooKeeper Protocol for an Update Workload.** *The figure shows the sequence of steps when the client interacts with the ZooKeeper cluster. The workload updates a value. The client prints to stdout once the update request is acknowledged.*

sentation of the update protocol, as shown in Figure 5.4.

First, the client contacts the leader in the ZooKeeper cluster. The leader receives the request and orchestrates the atomic broadcast protocol among its followers as shown by `send` and `recv` operations and careful updates to the file system (`write` and `fdatasync` on a log file that holds user data). Finally, after ensuring that the updated data is carefully replicated and persisted, the client is acknowledged. At this point, it is guaranteed that the data will be consistent and durable.

Note that each node runs multiple threads, and the figure shows the observed order of events when the traces were collected. If arbitrary delays were introduced, the order may or may not change, but this observed order is one schedule among all such possible schedules.

We reiterate here that PACE captures crash states that occur due to a correlated failure where all replicas fail together. PACE is not intended to reason about partial crashes where only a subset of replicas crash.

Globally Reachable Prefixes.

Assume that all nodes shown in Figure 5.4 start with persistent state X_ϕ where X is the node identifier with L for leader, C for client, and so forth. M_{XYi} is the ith message sent by X to Y. All operations that affect persistent state are annotated with the persistent state to which the node transitions by performing that operation. For example, the leader transitions to state L_1 after the first `write` to a file. The total set of global persistent states is the cross product of all local persistent states. Precisely, the total set is the cross product of the sets $\{C_\phi, C_1\}$, $\{L_\phi, L_1, L_2, L_3, L_4\}$, $\{P_\phi, P_1, P_2, P_3, P_4\}$ and $\{Q_\phi, Q_1, Q_2, Q_3, Q_4\}$. However, some of the global states in this resultant set cannot occur in the distributed execution. For example, $< C_\phi, L_2, P_2, Q_1 >$ is an inconsistent cut and cannot occur as a global state since it is not possible for Q to receive M_{PQ3} before P reaches P_3 and then sends M_{PQ3}.

We refer to a global state that is reachable in this trace as a globally reachable persistent prefix or simply *globally reachable prefix*. We call this a prefix as it is a prefix of the file-system operations within each node.

Previous work [181] has developed tools to uncover single-machine crash vulnerabilities. Such tools trace only file-system related system calls and do not trace network operations. Hence, they cannot capture dependencies across different nodes in a distributed system. Such tools cannot be directly applied to distributed systems; if applied, they may generate states that may not actually occur in a distributed execution and thus can

report spurious vulnerabilities. On the other hand, PACE captures all cross node dependencies and so generates only states that can occur in a distributed execution.

File-system Persistence Models.

Generating globally reachable prefixes does not require any knowledge about how a particular file system persists operations. As we discussed, file systems exhibit important behaviors with respect to how operations are persisted. We borrow the idea of *abstract persistence model (APM)* from our previous work [181] to model the file system used by each node.

An APM specifies all constraints on the atomicity and ordering of file-system operations for a given file system, thus defining which crash states are possible. For example, in an APM that specifies the ext2 file system, appends to a file can be reordered and the `rename` operation can be split into smaller operations such as deleting the source directory entry and creating the target directory entry. In contrast, in the ext3 (data-journaling) APM, appends to a file cannot be reordered, and the `rename` operation cannot be split into smaller operations. An APM for a new file system can be easily derived using the *block order breaker* (BOB) tool [181].

PACE considers all consistent cuts in the execution to find globally reachable prefixes. On each such globally reachable prefix, PACE applies the APM (that specifies what file-system specific crash states are possible) to produce more states. The default APM used by PACE has few restrictions on the possible crash states. Intuitively, our default APM models a file system that provides the least guarantees when crashes occur but is still `POSIX` compliant. For simplicity, we refer to file-system related system calls issued by the application as *logical operations* and the smaller operations into which each logical operation is broken down as *micro operations*. We now describe our default APM.

Atomicity of operations. Applications may require a single logical oper-

ation such as *append* or *overwrite* to be atomically persisted for correctness. In the default APM used by PACE, all logical operations are broken into the following micro operations: *write_block*, *change_size*, *create_dir_entry*, and *delete_dir_entry*. For example, a logical truncate of a file will be broken into *change_size* followed by *write_block(random)* followed by *write_block(zeroes)*. Similarly, a rename will be broken into *delete_dir_entry(dest) + truncate if last link* followed by *create_dir_entry(dest)* followed by *delete_dir_entry(src)*. Overwrites, truncates, and appends are split into micro operations aligned at the block boundary or simply into three micro operations. PACE can generate crash states corresponding to different intermediate states of the logical operation.

Ordering between operations. Applications may require that a logical operation *A* be persisted before another logical operation *B* for correctness. To reorder operations, PACE considers each pair of operations (*A*, *B*) and applies all operations from the beginning of the trace until *B* except for *A*. This reordering produces a state corresponding to the situation where the node crashes after all operations up to *B* have been persisted but *A* is still not persisted. The ordering constraint for our default APM is as follows: all operations followed by an *fsync* on a file or directory *F* are ordered after the operations on *F* that precede the *fsync*.

We now describe how applying an APM produces more states on a *single* machine. Consider the ZooKeeper protocol in which $< C_{\phi}, L_1, P_2, Q_{\phi} >$ is a globally reachable prefix. P has moved to P_2 by applying two `write` operations starting from its initial state P_{ϕ}. On applying the default APM onto the above prefix, PACE recognizes that on node P it is possible for the second write to reach the disk before the first one (by considering different ordering between two operations). Hence, it can *reorder* the first write after the second write on P. This resultant state is different from the prefix. In this resultant state, after recovery, P will see a file-system state where the second write to the log is persisted, but effects of the first

write are missing. If there were an `fsync` or `fdatasync` after the first write, then the default APM cannot and will not reorder the two write operations. This reordering is within a *single* node; similar reorderings can be exercised on all nodes.

Depending on the APM specification, logical operations can be partially persisted or reordered or both at each node in the system. Intuitively, applying an APM on a global prefix *relaxes* its constraints. This relaxation allows the APM to partially persist logical operations (atomicity) or reorder logical operations with one another (ordering). We refer to the relaxations allowed by an APM as *APM-allowed relaxations* or simply *APM relaxations*. For simplicity, we refer to this process of relaxing the constraints (by reordering and partially persisting operations) as applying that particular relaxation.

PACE can be configured with any APM. We find the most vulnerabilities with our default and ext2 APMs. We also report the vulnerabilities when PACE is configured with APMs of other commonly used file systems.

5.2.3 Protocol-Aware Exploration

While applying relaxations on a single node results in many persistent states for that node, PACE needs to consider applying different relaxations across every combination of nodes to find vulnerabilities. As a consequence, there are several choices for how PACE can apply relaxations. Consider a five node cluster and assume that n relaxations are possible in one node. Then, assuming there are no cross node dependencies, there are $\binom{5}{1} * n + \binom{5}{2} * n^2 + \binom{5}{3} * n^3 + \binom{5}{4} * n^4 + \binom{5}{5} * n^5$ ways of combining the relaxations across nodes. Even for a moderate n such as 20, there are close to 4 million states. A brute-force approach would explore all such states. We now explain how PACE prunes this space by using knowledge about the distributed protocols (such as agreement and leader election) employed by a system.

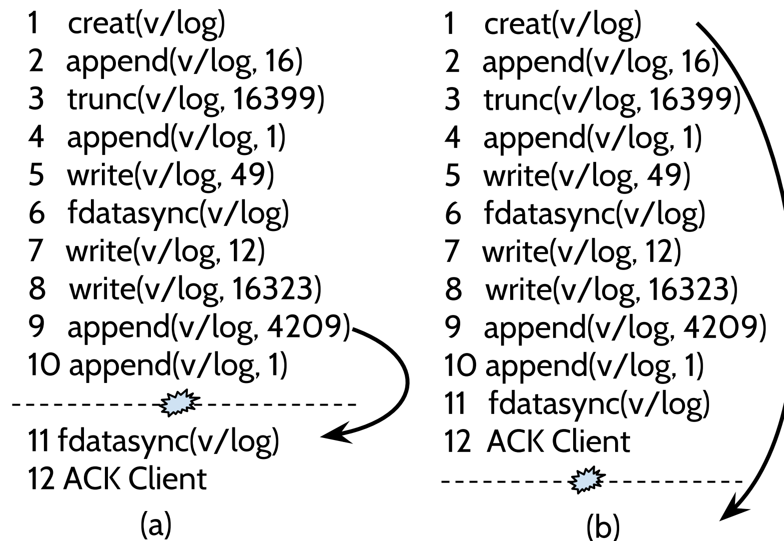


Figure 5.5: **Local File-system Update Protocol on a Single ZooKeeper Node.** *The figure shows the sequence of file-system operations on a single ZooKeeper node. Operations 1 through 6 happen on node initialization and operations 7 through 12 when the client starts interacting. Several operations that happen on initialization are not shown for clarity. (a) and (b) show two different crash scenarios.*

Replicated State Machine Approaches

We use the same ZooKeeper traces shown in Figure 5.4 for this discussion. For simplicity, we assume that there are odd number of nodes in the system.

ZooKeeper implements an atomic broadcast protocol which is required to run a replicated state machine (RSM) [112, 175, 205]. There are various paradigms to implement an RSM some of which include Paxos [130], Raft [175], and atomic broadcast [66]. Google's Chubby [44] implements a Paxos-like algorithm and LogCabin [144] implements Raft. An RSM system as a whole should continue to make progress as long as a *majority* of the nodes are operational and can communicate with each other and the clients [175].

Figure 5.5(a) shows the file-system operations on a single ZooKeeper node; network operations are not shown for clarity. The tenth operation appends one byte to the log to denote the commit of a transaction after which the file is forced to disk by the `fdatasync` call. It is possible for the tenth operation to reach the disk *before* the ninth operation and a crash can happen at this exact point before the `fdatasync` call. After this crash and subsequent restart, ZooKeeper would fail to start as it detects a checksum mismatch for the data written, and the node becomes unusable. The same reordering can happen on all nodes, rendering the entire cluster unusable.

In the simple case where this reordering happens on only one node, even though that single node would fail to start, the other two nodes still constitute a *majority* and so can elect a leader and make progress. PACE uses this knowledge about the protocol to eliminate testing cases where a reordering happens on only one node. Also, it is unnecessary to apply the relaxation on all three nodes as the cluster can become unavailable even when the relaxation is applied on just a majority (any two) of the nodes.

As another example, consider the same protocol but with a different crash that happens after the client is acknowledged, as shown in Figure 5.5(b). Once acknowledged, ZooKeeper guarantees that the data is replicated and persisted to disk on a majority of nodes. The directory entry for the log file has to be persisted explicitly by performing an `fsync` on the parent directory [5, 181] to ensure that the log file is present on disk even after a crash. However, ZooKeeper does not `fsync` the parent directory, and so it is possible for the log file to go missing after a crash. On a single node, if the log file is lost, it does not lead to user-visible global data loss as the majority still has the log file. Similar to the unavailability case, a global data loss can happen if the same reordering happens on a majority of nodes even if the data exists on one other node where this reordering did not happen.

Thus, we observe that in any RSM system, it is required that a particular APM relaxation is applied on at least a majority of nodes for a vulnerability to be exposed globally. Also, it is unnecessary to apply an APM relaxation on all possible majority choices; for example, in a system with five nodes, applying a relaxation on three, four, or five nodes (all of which represent a majority) will expose the same vulnerability. This knowledge is not system-specific but rather protocol-specific.

System-independent. LogCabin is a system similar to ZooKeeper that provides a configuration store on top of the consensus module but uses the Raft protocol to implement an RSM. When applying a particular APM relaxation, LogCabin can lose data. For this data loss vulnerability to be exposed, the relaxation has to be applied on at least a majority of the nodes. This observation is not specific to a particular system; rather, it holds true across ZooKeeper and LogCabin because both systems are RSM protocol implementations.

Using our observation, we derive the following rule that helps Pace eliminate a range of states: *For any RSM system with* N *replicas, check only states that would result when a particular APM relaxation is applied on an exact majority (where exactly* $\lfloor n/2 \rfloor + 1$ *servers are chosen from* n*) of the nodes*. Note that there are $\binom{n}{\lfloor n/2 \rfloor + 1}$ ways of choosing the exact majority.

We note that the pruning rule does not guarantee finding all vulnerabilities. It works because it makes an important assumption: the base consensus protocol is implemented correctly. Pace is not intended to catch bugs in consensus protocol implementations.

We now make a further observation about RSM protocols that can further reduce the state space. Consider the data loss vulnerability shown in Figure 5.5(b). Surprisingly, sometimes a global data loss may *not* be exposed even when the reordering happens on a majority. To see why consider that the current leader (L) and the first follower (P) lose the log file as the *creat* operation is not persisted before the crash. In this case,

the majority has lost the file. On recovery, the possibility of global data loss depends on *who is elected as the leader the next time*. Specifically, the data will be lost, if either L or P is elected as the new leader. On the other hand, if the second follower Q is elected as the leader, then the data will not be lost. In effect, the data will be lost if a node that lost its local data becomes the leader the next time, irrespective of the presence of the same data on other nodes.

In Raft, on detecting an inconsistency, the followers are forced to duplicate the leader's log (i.e., the log entries flow only outward from the leader) [175]. This enforcement is required to satisfy safety properties of Raft. While ZooKeeper's atomic broadcast (ZAB) does not explicitly specify if the log entries only flow outward from the leader, our experiments show that this is the case. Previous work also supports our observation [175].

This brings a question that counters our observation: *Why not apply the relaxation on any one node and make it the leader during recovery*? Consider the reordering shown in Figure 5.5(b). If this reordering happens on one node, that node will lose the log; it is *not* possible for this node to be elected the leader as other nodes would notice that this node has missing log entries and not vote for it. If this node is not elected the leader, then local data loss would not result in global data loss.

In contrast, if the log is lost on two nodes, the two nodes still constitute a majority, and so one of them can become the leader and therefore override the data on the third node causing a global data loss. However, it is possible for the third node Q, where the data was not lost, to become the leader and so hide the global data loss.

Given this information, we observe that it is required only to check states that result from applying a particular APM relaxation on *any one* exact majority of the nodes. In a cluster of five nodes, there are $\binom{5}{3} = 10$ ways of choosing an exact majority, and it is enough to check any one

combination from the ten. To effectively test if a global vulnerability can be exposed, we strive to enforce the following: *when the cluster recovers from a crashed state, if possible, the leader should be elected from the set of nodes where the APM relaxation was applied*. Sometimes the system may constrain us from enforcing this; however, if possible, we enforce it automatically to drive the system into vulnerable situations.

From the two observations, we arrive at two simple, system-independent, and protocol-aware exploration rules employed by PACE to prune the state space and effectively search for undesired behaviors:

- **R1**: *For any RSM system with* N *servers where followers duplicate leader's log, generate states that would result if a particular APM relaxation is applied on any exact majority of the servers.*
- **R2**: *For all states generated using R1, if possible, enforce that the leader is elected from exact majority in which the APM relaxation was applied.*

Since we did not see popular practical systems that use RSM approaches where log entries can flow in both directions like in Viewstamped replication [140, 171] or where there can be multiple proposers at the same time like in Paxos, we have not listed the rules for them.

Other Replication Schemes

PACE also handles replicated systems that do not use RSM approaches: Redis, Kafka, and MongoDB. Applications belonging to this category do not strictly require a majority for electing a leader and committing transactions. For example, in Redis' default configuration, the master is fixed and cannot be automatically re-elected by a majority of slaves if the master fails. Moreover, it is possible for the master to make progress without the slaves. Similarly, Kafka maintains a metadata structure called the *in-sync replicas*, and any node in this set can become the leader without consent from the majority.

Systems belonging to this category typically force slaves to sync data from the master. Hence, any problem in the master can easily propagate to the slaves. This hints that applying APM relaxations on the master is necessary. Next, since our workloads ensure that the data is synchronously replicated to all nodes, it is unacceptable to read stale data from the slaves once an acknowledgment is received. This hints that applying APM relaxations on any slave and subsequent reads from the slave can expose the stale data problem. Since systems of this type can make progress even if one node is up, we need to apply APM relaxations on all the nodes to expose cluster unavailability vulnerabilities.

For applications of this type, Pace uses a combination of the following rules to explore the state space:

- **R3**: *Generate states that result when a particular relaxation is applied on the master.*
- **R4**: *Generate states that result when a particular relaxation is applied on any one slave.*
- **R5**: *Generate states that result when a particular relaxation is applied on all nodes at the same time.*

In Redis, we use R3 and R4 but *not* R5: we use R3 to impose APM relaxations only on the master because the cluster can become unavailable for writes if only the master fails; we use R4 as reads can go to slaves. Similarly, in Kafka, we use R3 and R5 and *not* R4: we do not use R4 because all reads and writes go only through the leader; we use R5 to test states where the entire cluster can become unavailable because the cluster will be usable even if one node functions. MongoDB can be configured in many ways. We configure it much like an RSM system where it requires a majority for leader election and writes; hence, we use R1 and R2.

Examining a new distributed system with Pace requires developers to only understand whether the system implements a replicated state machine or not and how the master election works. Once this is known, Pace can be easily configured with the appropriate set of pruning rules. We believe that Pace can be readily helpful to developers given that they already know their system's protocols. We reiterate that the pruning rules do not guarantee finding all vulnerabilities; rather, they provide a set of guidelines to quickly search for problems. In the worst case, if no properties are known about a protocol, Pace can work in brute-force mode to find vulnerabilities.

Effectiveness of Pruning

To demonstrate the effectiveness of our pruning rules, we explored crash states of Redis and LogCabin with Pace and the brute-force approach. In Redis, for a simple workload on a three node cluster, brute-force needs to check 11,351 states, whereas Pace only needs to check 1009 states. While exploring 11× fewer states, Pace found the same three vulnerabilities as the brute-force approach. In LogCabin, Pace discovers two vulnerabilities, checking 27,713 states in eight hours; the brute-force approach did not find any new vulnerabilities after running for over a week and exploring nearly 900,000 states. The reduction would be more pronounced as the number of nodes in a system increases.

5.2.4 Limitations and Caveats

We first note that Pace is not intended to catch bugs in distributed consensus protocols. Specifically, it does not exercise reordering of network messages to explore corner cases in consensus protocols; as we explain later in the related work chapter, distributed model checkers attack this problem. Pace's intention is to examine the interaction of global crash re-

System	Tested Version
Redis	v3.0.4
ZooKeeper	v3.4.8
LogCabin	v1.0.0
etcd	v2.3.0
RethinkDB	v2.2.5
MongoDB	v3.0.11
iNexus	v0.13
Kafka	v0.9.0

Table 5.1: **System Versions.** *The table shows the versions of the systems that we tested with* PACE.

covery protocols and the nuances in local storage protocols (introduced by each replica's local file system), in the presence of correlated crashes.

Second, PACE is *not complete* – it can miss vulnerabilities. Specifically, PACE exercises only one and the same reordering at a time across the set of nodes. For instance, consider two reorderings r_i and r_j. It is possible that no vulnerability is seen if r_i or r_j is applied individually on two nodes. But when r_i is applied on one node and r_j on the other, then it may lead to a vulnerability. PACE would miss such vulnerabilities. Note that if r_i and r_j can both individually cause a vulnerability, then PACE would catch both of them individually. This is a limitation in implementation and not a fundamental one. There is *no* similar limitation with partially persisting operations (i.e., PACE can partially persist different operations across nodes).

5.3 Vulnerabilities Study

We studied eight widely used distributed systems spanning different domains including database caches (Redis), configuration stores (ZooKeeper,

LogCabin, etcd), real-time databases (RethinkDB), document stores (MongoDB), key-value stores (iNexus), and message queues (Kafka). We tested MongoDB with two storage engines: WiredTiger [164] (MongoDB-WT) and RocksDB [199] (MongoDB-R). Table 5.1 shows the versions of the systems tested. Pace found **26** unique vulnerabilities across the eight systems.

We first describe the workloads and checkers we used to detect vulnerabilities (§5.3.1). We then present a few example protocols and vulnerabilities to give an intuition of our methodology and the types of vulnerabilities discovered (§5.3.3). We then answer three important questions: Are there common patterns in file-system requirements (§5.3.4)? What are the consequences of the vulnerabilities discovered by Pace (§5.3.5)? How many vulnerabilities are exposed on real file systems (§5.3.6)? We finally describe our experience with reporting the vulnerabilities to application developers (§5.3.7).

5.3.1 Workloads and Checkers

Most systems have configuration options that change user-level guarantees. We configured each system to provide the highest level of safety guarantees possible. When guarantees provided are unclear, our checkers check for typical user expectations; for example, data acknowledged as committed should not be lost in any case, or the cluster should be available after recovering from crashes. Even though some applications do not explicitly guarantee such properties, we believe it is reasonable to test for such common expectations.

To test a system, we first construct a workload. Our workloads are not specifically crafted to expose vulnerabilities, but rather are very natural and simple. Our workloads insert new data or update existing data and record the acknowledgment from the cluster. They are usually about 30-40 LOC.

To check each crash state, we implement a checker. The checker is con-

System	Configuration	Workload	Checker
Redis	*appendfsync=always*, *min-slaves-to-write=2* and *wait*	update existing	old and new data (master and slave), *check-aof*, *check-dump*
ZooKeeper	Default	update existing	old and new data
LogCabin	Default	update existing	old and new data
etcd	Default	update existing	old and new data
RethinkDB	*durability=hard*, *writeack=majority*	update existing, insert new	old and new data
MongoDB	*W=3*, *journal=true*	update existing	old and new data
iNexus	Default	update existing, insert new	old and new data
Kafka	*flush.interval.msgs=1*, *min in-sync replicas=3*, *DirtyElection=False*	create topic, insert message	topic and message

Table 5.2: **Configurations, Workloads, and Checkers.** *The table shows the configuration, workloads and checkers for each system. We configured all systems with three nodes. The configuration settings ensure data is synchronously replicated and flushed to disk.*

ceptually simple; it starts the cluster with the crash state produced by PACE and checks for correctness by reading the data updated by the workload. If the data is lost, corrupted, or not retrievable, the checker flags the crash state incorrect. Further, our checkers invoke recovery tools mentioned in applications' documentation if an undesired output is observed. If the problem is fixed after invoking the recovery tool, then it is *not* reported as a vulnerability. Our checkers are about 100 LOC. Table 5.2 shows the configurations (that achieve the strongest safety guarantees), workloads,

```
## Workload ##
# Start cluster
# Insert new data
zk =
client(hosts=server_ips)
zk.set("/mykey",
"newvalue")
pace.acknowledged = True
# Stop cluster

## Checker ##
# Start cluster
# Check for data
retry_policy = retry(max_tries = r, delay = d,
backoff = b)
zk = client(hosts=server_ips, retry_policy)
ret, stat = zk.get("/mykey")
if request succeeded:
  if pace.acknowledged and ret == None:
    return 'data loss new commit'
  if pace.acknowledged and ret != 'newvalue':
    return 'corrupt'
  if not pace.acknowledged and ret == None:
    return 'data loss old commit'
else:
  return 'unavailable'
return 'correct'
# Stop cluster
```

Listing 5.1: **Workload and Checker.** *Simplified workload and checker for ZooKeeper.*

and checkers for all systems. Listing 5.1 shows the simplified pseudocode of the workload and the checker for ZooKeeper.

5.3.2 Vulnerability Accounting

A system has a crash vulnerability if a crash exposes a user-level guarantee violation. Counting such vulnerable places in the code is simple for single-machine applications. In a distributed system, multiple copies of

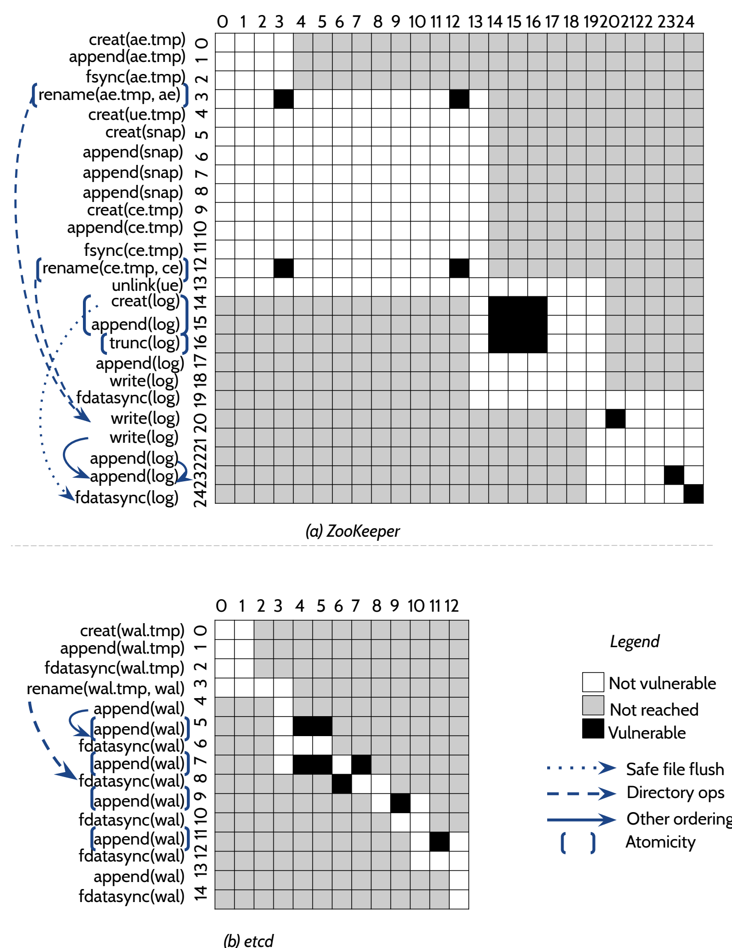


Figure 5.6: **Protocols and Vulnerabilities: ZooKeeper and etcd.** *(a) and (b) show protocols and vulnerabilities in ZooKeeper and etcd, respectively. States that are not vulnerable, that were not reached in the execution, and that are vulnerable are shown by white, grey, and black boxes, respectively. The annotations show how a particular state becomes vulnerable. In Zookeeper, box (24, 24) is vulnerable because both nodes crash after the final fdatasync but before the log creation is persisted. Atomicity vulnerabilities are shown with brackets enclosing the operations that need to be persisted atomically. The arrows show the ordering dependencies in the application protocol; if not satisfied, vulnerabilities are observed. Dotted, dashed, and solid arrows represent safe file flush, directory operation, and other ordering dependencies, respectively.*

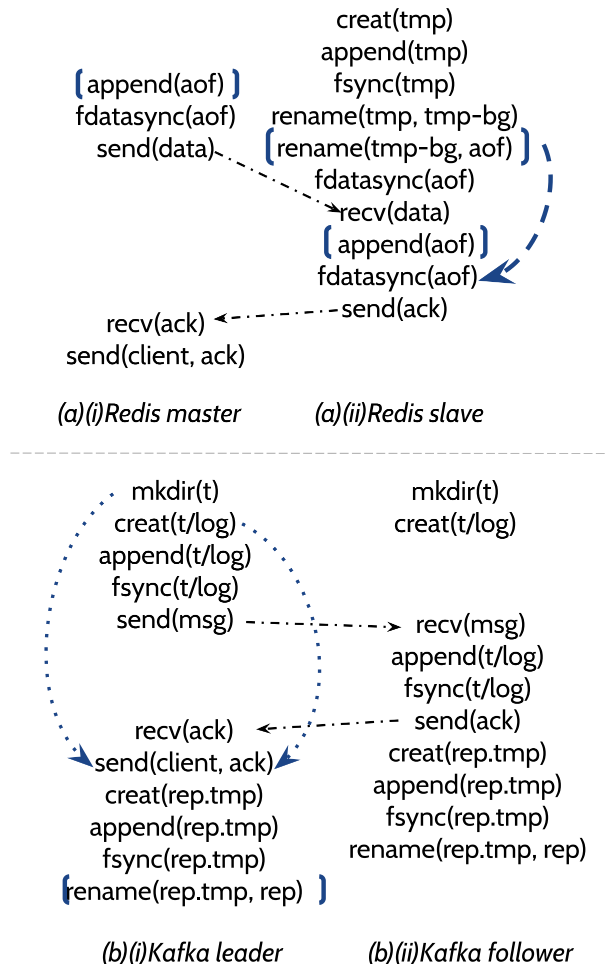


Figure 5.7: **Example Protocols and Vulnerabilities: Redis and Kafka.** *(a) and (b) show protocols and vulnerabilities in Redis and Kafka, respectively. Refer Figure 5.6 for legend.*

the same code execute and so Pace needs to be careful in how it counts *unique* vulnerabilities.

We count only unique combinations of states that expose a vulnerability. Consider a sequence *S1* that creates (C), appends (A), and renames (R) a file. Assume that a node will not start if it crashes after C, but before R. Assume there are three nodes in an RSM system and two crash after C,

but before R. In this case, the cluster can become unusable in four ways (C-C, CA-CA, C-CA, CA-C). We count all such instances as one vulnerability. If the third node crashes within this sequence, it will also be mapped onto the same vulnerability. If there is another different sequence *S2* that causes problems, a vulnerability could be exposed in many different ways as one node can crash within *S1* and another within *S2*. We associate all such combinations to two unique vulnerabilities, attributing to the atomicity of *S1* and *S2*.

PACE also associates each vulnerability with the application source code line using the stack trace information obtained during tracing. When many vulnerabilities map to the same source line, PACE considers that a single vulnerability. When we are unable to find the exact source lines for two different vulnerabilities, we count them as one. We note that our way of counting vulnerabilities results in a conservative estimate.

5.3.3 Example Protocols and Vulnerabilities

Figure 5.6 shows the protocols and vulnerabilities in ZooKeeper and etcd. Figure 5.7 shows the same for Redis and Kafka. RSM systems where vulnerabilities are exposed when APM relaxations are applied on a majority of nodes are represented using a grid. Figure 5.6(a) and 5.6(b) show the combinations of persistent states across two nodes in a three node ZooKeeper and etcd cluster, respectively. Operations that change persistent state are shown on the left (for one node) and the top (for the other node). A box *(i,j)* corresponds to a crash point where the first node crashes at operation *i* and the second at *j*. At each such crash point, PACE reorders other operations, or partially persists operations or both. A grey box denotes that the distributed execution did not reach that combination of states. A white box means that after applying all APM relaxations, PACE was unable to find a vulnerability. A black box denotes that when a specific relaxation (shown on the left) is applied, a vulnerability is exposed.

As shown in Figure 5.6(a), to maintain proposal information, ZooKeeper appends epoch numbers to temporary files and renames them. If the renames are not atomic or reordered after a later write, the cluster becomes unavailable. If a log file creation and a subsequent append of header metadata are not atomically persisted, then the nodes fail to start. Similarly, the immediate truncate after log creation has to be atomically persisted for correct startup. Writes and appends during transactions, if reordered, can also cause node startup failures. ZooKeeper can lose data as it does not `fsync` the parent directory when a log is created.

Figure 5.6(b) shows the protocol and vulnerabilities in etcd. etcd creates a temporary write-ahead log (WAL), appends some metadata, and renames it to create the final WAL. The WAL is appended, flushed, and then the client is acknowledged. We find that an etcd cluster becomes unavailable if crashes occur when the WAL is appended; the nodes fail to start if the appends to the WAL are reordered or not persisted atomically. Also, if the rename of the WAL is reordered, a global data loss is observed.

Non-RSM systems where vulnerabilities are exposed even when relaxations are applied on a single machine are shown using trace pairs. As shown in Figure 5.7(a), Redis uses an append-only file to store user data. The master appends to the file and sends the update to slaves. Slaves, on startup, rewrite and rename the append-only file. When the master sends new data, the slaves append it to their append-only file and sync it. After the slaves respond, the client is acknowledged. Data loss windows are seen if the rename of the append-only file is not atomic or reordered after the final fdatasync. When the append is not atomic on the master, a user-visible silent corruption is observed. Moreover, the corrupted data is propagated from the master to the slaves, overriding their correct data. The same append (which maps to the same source line) on the slave results in a window of silent corruption. The window closes eventually

since the slaves sync the data from the master on startup.

Figure 5.7(b) shows the update protocol of Kafka. Kafka creates a log file in the topic directory to store messages. When a message is added, the leader appends the message and flushes the log. It then contacts the followers which perform the same operation and respond. After acknowledging the client, the replication offset (that tracks which messages are replicated to other brokers) is appended to a temporary file, flushed, and renamed to the replication-offset-checkpoint file. The log can be lost after a crash because its parent directory is not flushed after the log creation. If the log is lost on the master, then the data is globally lost since the master instructs the slaves also to drop the messages in the log. Similarly, Kafka can lose a message topic altogether since the parent directory of the topic directory is not explicitly flushed.

We observe that some systems (e.g., Redis, Kafka) do not effectively use redundancy as a source of recovery. For instance, in these systems, a local problem (such as a local corruption or data loss) which results due to a relaxation on a single node, can easily become a global vulnerability such as a user-visible silent corruption or data loss. In such situations, these systems miss opportunities to use other intact replicas to recover from the local problem. Moreover, such local problems are propagated to other intact replicas, overriding their correct data.

5.3.4 Patterns in File-system Requirements

Table 5.3 shows file-system requirements across systems. We group the results into three patterns:

Inter-Syscall Atomicity. ZooKeeper and LogCabin require inter system call atomicity (multiple system calls need to be atomically persisted). In both these systems, when a new log file is initialized, the creat and the initial append of the log header need to be atomically persisted. If the log initialization is partially persisted, the cluster becomes unavailable.

System	FS Requirements							Unique vulnerabilities
	Inter-syscall atomicity	Atomicity			Ordering			
		Appends and truncates	Rename (dest link absent)	Rename (dest link exists)	Safe file flush	Directory ops	Other	
Redis		1		1		1		**3**
ZooKeeper	1	1		1	1	1	1	**6**
LogCabin	1		1					**2**
etcd		1				1	1	**3**
RethinkDB								
MongoDB-WT				1				**1**
MongoDB-R		1		1	1	1	1	**5**
iNexus				1		1	1	**3**
Kafka				1	2			**3**
Total	2	4	1	6	4	5	4	**26**

Table 5.3: **Vulnerabilities: File-System Requirements.** *The table shows the unique vulnerabilities categorized by file-system requirements.*

Vulnerabilities due to inter system call atomicity requirements can occur on all file systems irrespective of how they persist operations.

Atomicity within System calls. We find that seven systems require system calls to be atomically persisted. Eleven unique vulnerabilities are observed when system calls are not persisted atomically. Six out of the eleven vulnerabilities are dependent on atomic replace by rename (destination link already exists), one on atomic create by rename (destination link does not exist), and four on atomic truncates or appends. Four sys-

System	Consequences								Unique vulnerabilities
	Silent Corruption	Data Loss		Unavailable		Window			
		Old commit	New commit	Metadata corruption	User data corruption	Corruption	Data loss old commit	Data loss new commit	
Redis	1					1	1	1	**3**
ZooKeeper			1	4	1				**6**
LogCabin		1		1			1		**2**
etcd			1	1	2				**3**
RethinkDB									
MongoDB-WT				1					**1**
MongoDB-R			3	3					**5**
iNexus		1	1	2					**3**
Kafka			3						**3**
Total	1	2	9	12	3	1	2	1	**26**

Table 5.4: **Vulnerabilities: Consequences.** *The table shows the unique vulnerabilities categorized by user-visible consequences.*

tems require appends or truncates to be atomic. Redis, ZooKeeper, and etcd can handle appended portions filled with zeros but not garbage.

Ordering between System calls. Six systems expect system calls to be persisted in order. Kafka and ZooKeeper suffer from data loss since they expect the safe file flush property from the file system. To persist a file's directory entry, the parent directory has to be explicitly flushed to avoid such vulnerabilities. We found that reordering directory operations can cause vulnerabilities. We found that five systems depend on ordered renames: Redis exhibits a data loss window, etcd permanently loses data,

ZooKeeper, MongoDB-R, and iNexus fail to start. Four systems require other operations (appends and writes) to be ordered for correct behavior.

5.3.5 Vulnerability Consequences

Table 5.4 shows the vulnerability consequences. We find that all vulnerabilities have severe consequences like silent corruption, data loss, or cluster unavailability. Redis silently returns and propagates corrupted data from the master to slaves even if slaves have correct older version of data. Redis also has a silent corruption window when reads are performed on slaves. While only one system silently corrupts and propagates corrupted data, six out of eight systems are affected by permanent data loss. Depending on the crash state, previously committed data can be lost when new data is inserted, or the newly inserted data can be lost after acknowledgment. Redis exhibits a data loss window that is exposed when reads are performed on the slaves. As slaves continuously sync data from the master, the window eventually closes.

Cluster unavailability occurs when nodes fail to start due to corrupted application data or metadata. ZooKeeper and etcd fail to start if CRC checksums mismatch in user data. MongoDB-WT fails to start if the *turtle* file is missing and MongoDB-R fails to start if the *sstable* file is missing or there is a mismatch in the *current* and *manifest* files. LogCabin and iNexus skip log entries when checksums do not match but fail to start if metadata is corrupted. LogCabin fails to start when an unexpected segment metadata version is found. Similarly, ZooKeeper fails to start on unexpected epoch values. While some of these scenarios can be fixed by expert application users, the process is intricate and error prone.

We note that the vulnerabilities are specific to our simple workloads and all vulnerabilities reported by Pace have harmful consequences. More complex workloads and checkers that assert more subtle invariants are bound to find more vulnerabilities.

	ext2	ext3-w	ext3-o	ext4-o	ext3-j	btrfs
Redis	3	1				1
ZooKeeper	6	3	1	1	1	3
LogCabin	2	1	1	1	1	1
etcd	3	2				
MongoDB-WT	1					
MongoDB-R	5	2	2	2		3
iNexus	2		1	1		2
Kafka	3					
Total	26	9	5	5	2	10

Table 5.5: **Vulnerabilities on Real File Systems.** *The table shows the number of vulnerabilities on commonly used file systems.*

5.3.6 Impact on Real File Systems

We configured Pace with APMs of real file systems. Table 5.5 shows the vulnerabilities on each file system. We observe that many vulnerabilities can occur on all examined file systems. Only two vulnerabilities are observed in ext3-j (data-journaling) as all operations are persisted in order. All vulnerabilities that occur on our default APM are also exposed on ext2. Systems are vulnerable even on Linux's default file system (ext4 ordered mode). Many of the vulnerabilities are exposed on btrfs as it reorders directory operations. In summary, the vulnerabilities are exposed on many current file systems on which distributed storage systems run today.

5.3.7 Confirmation of Problems Found

We reported 18 of the discovered vulnerabilities to application developers. We confirmed that the reported issues cause serious problems (such as data loss and unavailability) to users of the system. Seven out of the 18 reported issues were assigned to developers and fixed [82–84, 143, 200,

201]. Another five issues have been acknowledged or assigned to developers. Out of this five, two in Kafka were already known [121]. Other issues are still open and under consideration. We found that distributed storage system developers, in general, are responsive to such bug reports for two reasons. First, we believe developers consider crashes very important in distributed systems compared to single-machine applications. Second, the discovered vulnerabilities due to crashes affect their users directly (for example, data loss and cluster unavailability).

We found that users and random-crash testing have also occasionally encountered the same vulnerabilities that were systematically discovered by Pace. However, Pace diagnoses the underlying root cause and provides information of the problematic source code line, easing the process of fixing these vulnerabilities.

5.3.8 Discussion

We now list a few high-level lessons that we learned through our study of vulnerabilities.

First, we find that redundancy by replication is not the panacea for constructing reliable storage systems. A common expectation in distributed systems is that if a node loses its data (for example, due to file-system crash behaviors as in our study), then the node can be fixed using the redundant copies of data on the other nodes. However, we find that the opposite was true in many systems: the problematic node spreads its copy to the other nodes, causing spread of corruption or data loss. For example, Redis and Kafka can propagate corrupted data and data loss to slaves, respectively. We believe replication protocols and local storage protocols should be designed in tandem to avoid such undesired behaviors.

Second, system designers need to be careful about two problems when embracing *layered* software. First, the reliability of the entire system depends on individual components. MongoDB's reliability varies depend-

ing on the storage engine (WiredTiger or RocksDB). Second, separate well-tested components when integrated can bring unexpected problems. In the version of MongoDB we tested, we found that correct options are not passed from upper layers to RocksDB, resulting in a data loss. Similarly, iNexus uses a modified version of LevelDB which does not flush writes to disk when transactions commit. Applications need to clearly understand the guarantees provided by components when using them.

Third, we find that a few systems are overly cautious in how they update file-system state. LogCabin flushes files and directories after every operation. Though this avoids many reordering vulnerabilities, it does not fix atomicity vulnerabilities. Issuing `fsync` at various places does not completely avoid reliability problems. Also, the implication of too much caution is clear: low performance. While this approach is reasonable for configuration stores, key-value stores need a better way to achieve the same effect without compromising performance.

Next, we note that sometimes the programming environment may constrain applications from doing the right thing, leading to vulnerabilities. For example, consider the safe file flush and directory-operation reordering vulnerabilities in ZooKeeper and Kafka. These vulnerabilities arise because these systems are written in Java in which `fsync` cannot be readily issued on directories.

Finally, all modern distributed storage system run on top of a variety of file systems that provide different crash guarantees. We advocate that distributed storage systems should understand and document on which file systems their protocols work correctly to help practitioners make conscious deployment decisions.

5.4 Solving the Problems found by PACE

In this section, we discuss how the problems found by Pace can be solved. We believe there are two different approaches to solve these problems. First, the unintuitive crash states can be avoided locally on each node; this solution can be realized by running on file systems that provide clear guarantees on crashes or by fixing the update protocol of the application (e.g., by issuing `fsync` calls at appropriate places). This approach may reduce common-case performance. Second, applications can continue running atop file systems that may produce unintuitive states, and rely upon recovery from redundant copies to fix local problems when they arise. We believe this approach is suitable when applications cannot control what file systems they will be run on.

5.4.1 Local Hardening

We consider the 26 unique vulnerabilities found by Pace and analyze how each of them can be solved using a local approach. First, seven vulnerabilities occur due to non-atomic renames and thus possess a practical concern only when the applications are run on file systems that do not provide such guarantees (e.g., ext2). However, given that most modern file systems such as ext3, ext4, and btrfs ensure rename atomicity, we believe these vulnerabilities do not need any fix.

Next, 13 vulnerabilities in our study are caused by reordering. These reordering vulnerabilities can be fixed by carefully issuing `fsync` at correct places in the local update protocol of the systems; however, such fixes can impact the common-case performance. Another option would be to run atop file systems that aim to improve crash consistency by providing ordered updates. Following our work on finding crash vulnerabilities in distributed systems, researchers have designed new file systems such as CCFS that provide ordered updates with high performance [180].

Some of these vulnerabilities will thus be masked without forgoing too much performance if distributed systems run CCFS at each of the replicas. However, some reordering vulnerabilities may not be fixed upon CCFS and `fsync` calls may still be required if applications want durability. Thus, fixing such durability vulnerabilities will inherently impact performance.

Of the remaining six vulnerabilities, two are caused by inter-syscall atomicity, i.e., the system expects two systems calls to be atomically persisted. These vulnerabilities can be exposed on all current file systems and CCFS cannot help mask these problems. While a transactional interface that allows multiple system calls to be atomically persisted can help, such an interface is far from reality in current commodity file systems. However, we found that both of these vulnerabilities can be handled gracefully by fixing the application recovery code. Specifically, when the node recovers, the recovery code should expect that the state may contain the first operation but not the second. This was a fairly straightforward fix and has been implemented in LogCabin [143].

The remaining four vulnerabilities are caused by multi-block appends or truncates that the applications expect to be atomic. These vulnerabilities can be exposed on all current file systems and CCFS cannot mask them. In three instances of these vulnerabilities, the application already uses checksums to detect the non-atomic append. However, upon a detection, they take an undesirable step such as crashing the node, leading to unavailability. Such instances can be better handled by truncating the corrupted portions and continuing. In the remaining one instance, the system does not use checksums to detect the problem and further propagates the corrupted data to other nodes. To fix this, the system must use checksums first and then take the same steps described above.

5.4.2 Handling Using Distributed Redundancy

In this second option, systems do not rely heavily upon on the correctness of the local file system. Instead, they use the inherent redundancy to fix local problems that arise when crashes occur. We believe this solution is suitable in scenarios where the application does not have control on which file systems it will run on. We believe these scenarios are important to address because developers of distributed systems systems do not (and cannot) control how practitioners deploy these systems in the real world. Further, while developers may be confident that their system may work on many existing file systems, they cannot be sure about guarantees that may be broken in future file systems.

In the above scenarios, the local file system may be weak and so may not provide some common guarantees (e.g., safe file flush, without which an explicit `fsync` on the directory is required to persist file creations and deletions). Upon such file systems, a node may lose its data locally on a crash. For example, in our study, we found a Kafka node can lose its data locally if the file system does not provide the safe file flush guarantee. However, Kafka (similar to other many other systems in our study) does not fix this local data loss using the redundant copies; on the contrary, it propagates this local data loss to the followers by allowing the node that lost its data to become the leader. An ideal fix for this problem would be to preclude such a node from becoming the leader and so when it becomes a follower, the node can fix its data from the leader's copy.

5.5 Summary and Conclusions

In this chapter, we discussed how crash behaviors of local file systems influence the correctness of distributed update protocols. We presented Pace, a tool that can effectively search for correlated crash vulnerabilities by pruning the search space. We studied eight popular distributed stor-

age systems using Pace and exposed many serious vulnerabilities, many of which have been acknowledged and fixed by developers. Source code of Pace, workloads, checkers, and details of the discovered vulnerabilities are publicly available [6].

6
Related Work

In this chapter, we discuss how prior systems and research efforts relate to various aspects of the work presented in this dissertation. First, we discuss prior studies that show the prevalence of storage faults; we also discuss various past efforts to understand how different systems and layers (such as the local file system) react to storage faults (§6.1). Then, we describe other approaches used in distributed systems to handle storage faults (§6.2). Next, we discuss how the techniques used in Saucr are different from other related approaches (§6.3). We then discuss how research and practical efforts have attempted to test the reliability of distributed systems by building a variety of tools (§6.4); we show our work on Pace has some similar flavor to this body of work, but at the same time differs from past tools in important ways. Finally, we describe attempts to find crash vulnerabilities in single-machine applications (§6.5); we discuss how these tools cannot find the vulnerabilities that we find with Pace.

6.1 Studies on Storage Faults

6.1.1 Prevalence of Storage Faults

Our analysis of how RSM-based systems react to storage faults draws inspiration from many past fault-injection studies. Several studies have shown the prevalence of storage faults in hard disk drives [206] and SSDs [101,

154, 168, 207]. Further, studies have shown that cheap and near-line storage devices are more likely to be affected by storage faults [31, 32]. Given that many distributed deployments tend to use cheap storage hardware [70, 94], analyzing the effects of storage faults is important. These prior studies motivated us to analyze how RSM systems behave in the presence of storage faults.

6.1.2 File-system Behaviors to Storage Faults

Closely related to our work is the work on IRON file systems [103, 186]. In this work, the authors analyze how local file systems (such as ext3 and IBM JFS) react to partial storage faults. Our work is different from this effort: while their analysis concentrates on file systems, we look at a layer above such local file systems. From their study, the authors find that local file systems do not handle storage faults correctly in many scenarios and that they use illogically inconsistent policies to detect and recover from storage faults. These findings indicate that applications running atop local file systems are ultimately responsible for maintaining end-to-end data integrity. Similar results have been established by other related studies on how file systems react to storage faults [33, 242].

6.1.3 Application Behaviors to Storage Faults

Another body of work has studied how applications that run atop file systems react to storage faults. For instance, Subramanian et al. study how open-source DBMS systems react to storage corruptions [213]. Their study reveals that systems such as MySql are not robust to storage faults: the system may crash, lose data, or even return incorrect results. Our analysis of how RSM systems react to such faults is related but different: the authors in that work study a single-machine application, whereas we focus on distributed systems. The fundamental difference is that in

a single-machine setting, the application rarely has ways to recover from the fault; in contrast, in a distributed setting, data is inherently replicated, offering a way to recover from storage faults. Zhang et al., study the effects of storage corruptions on cloud-based synchronization services; the authors find that data can be lost even when there are copies spread across many user devices and the cloud service [240]. Our results are similar to these findings but in a much more critical class of systems.

6.1.4 Distributed-system Reactions to Storage Faults

More recently, in our own work (that is not a part of this dissertation), we have studied how popular distributed storage systems behave in the presence of storage faults [91, 92]. In this work, we discovered fundamental reasons why distributed systems are not resilient to storage faults. At a high level, similar to our analysis in §3.1, the prior work also found that distributed systems do not effectively use redundancy to recover from storage faults. However, the study did not uncover any safety or availability violations reported in §3.1; this is because the fault model in our previous study considers injecting only storage faults (precisely, a single storage fault on a single node at a time). In contrast, our fault model in Par considers crashes and network failures in addition to storage faults, exposing previously unknown safety and availability violations in RSM systems. To the best of our knowledge, the analysis presented in this thesis is the first to expose these violations.

6.2 Approaches to Handling Storage Faults

Our design of Ctrl in Chapter 3 builds upon several past efforts on tolerating storage faults such as data corruption and latent sector errors. Among these efforts, we note that there are two classes of approaches. In the first, the local storage layer (either the storage hardware itself or some layer

above it, such as the local file system) is made more reliable by using error correction and internal-redundancy techniques. In the second, the problems are handled at the distributed layer. We next discuss each of these approaches in turn.

6.2.1 More Reliable Local Storage

Many prior efforts have tried to improve the reliability of the local storage stacks, hardening them against storage faults. A few systems have explored techniques at the hardware level, for example, by employing different kinds of error detection and correction schemes [45, 141, 243].

Another body of work argues that the limited trust must be placed on the hardware in handling storage faults and makes the case that software should handle such problems. This body of work mainly focuses on developing techniques to improve the resiliency of local file systems. Early work on IRON file systems [186] shows how techniques such as redundancy, parity, etc., can be built into the local file system to tolerate partial storage faults without affecting performance. Researchers have applied similar ideas to other file systems such as Sun ZFS [241]. Other efforts have tried to improve the reliability of file systems to storage faults using isolation and recovery techniques [146, 217], or N-versioning techniques [34].

6.2.2 Handling at the Distributed Layer

In contrast to previous approaches that intend to contain storage faults completely in the storage layer, another body of work proposes to handle these problems at the distributed layer. This way of thinking about handling storage faults in distributed systems was initiated by early work on the Google File System [70, 94], which proposed to use inexpensive disks and to handle faults using redundancy at the distributed software

level. This approach, in addition to complying with the end-to-end principles [203], also frees the storage devices from offering stringent error detection and correction properties [42, 219]. This body of research to tolerate disk errors at the distributed level closely matches our views in our work on protocol-aware recovery presented in Chapter 3. However, our work is the first to devise recovery mechanisms for RSM systems.

More recently, the DIRECT system (that follows our work on Par) recovers from bit-level errors in distributed systems such as HDFS [220]. In this paper, the authors propose extending flash lifetime by allowing devices to expose higher bit error rates. Atop such unreliable devices, DIRECT uses the redundancy in distributed systems to recover from bit corruption errors. In doing so, DIRECT extends the lifetime of flash devices by utilizing these devices even after they begin exposing bit errors. Similar approaches that harden distributed file systems such as HDFS to storage faults have also been proposed in the past [223, 232]. This body of work has similar goals to our work on Par. While these ideas improve the reliability of distributed file systems, our work focuses on a more fundamental component, RSM systems, whose reliability is crucial to the correct functioning of many data-center systems (including distributed file systems).

Targeted Approaches in RSM Systems

While the above approaches intend to handle storage faults in distributed systems in general, a few previous efforts have described (however, only very briefly) how data corruption can be handled in RSM systems. Prior research describes two such ways [40, 50]: *MarkNonVoting* and *Reconfigure*. However, as we showed in Section §3.1, these approaches suffer from unavailability. Ctrl provides better availability than these solutions. Furthermore, the *MarkNonVoting* approach [50] can violate safety because important metainfo such as promises can be lost on a storage fault [231].

CTRL avoids such safety violations by storing two copies of metainfo on each node.

Generic Approaches in RSM Systems

Many generic approaches to handling practical faults in RSM systems other than crashes have been proposed. PASC [65] hardens systems to tolerate corruptions by maintaining two copies of the entire state on each node and assumes that both the copies will not be faulty at the same time. This approach does not work well for storage faults; having two copies of on-disk state incurs $2\times$ space overhead. Furthermore, in most cases, PASC crashes the node on a fault; such local strategies, as we showed in Section §3.1, can cause unavailability. XFT [142] is designed to tolerate non-crash faults. However, it can tolerate only a total of $\lfloor (N-1)/2 \rfloor$ crash and non-crash faults. Similarly, UpRight [59] has an upper bound on the total faults to remain safe and available.

CTRL differs from the generic approaches through its special focus on storage faults. This focus brings two main advantages. First, CTRL attributes faults at a fine granularity: while the generic approaches consider a node as faulty if any of its data is corrupted, CTRL considers faults at the granularity of individual data items. Second, because of such fine-granular fault treatment, CTRL can be available as long as a majority of nodes are up and at least one non-faulty copy of a data item exists even though portions of data on *all* nodes could be corrupted. CTRL cannot tolerate arbitrary non-crash faults [128] (e.g., memory errors). However, CTRL can augment the generic approaches: for example, a system can be hardened against memory faults using PASC while making it robust to storage faults using CTRL.

6.3 SAUCR Techniques

We now discuss how prior systems and research efforts relate to various aspects of our work on situation-aware updates and crash recovery.

6.3.1 Failure Detection and Reaction

The durability and availability properties of Saucr are dependent upon how fast it can detect failures as they are happening. A large body of work exists showing how to build fast and practical failure detectors [136–138]. Saucr's failure detection is also fast (as shown by our experiments) and is practical (it only uses heartbeats, a common mechanism readily available in many systems).

Previous failure detectors also strive to be reliable by avoiding false positives: cases where the detector may inform that a node has failed when, in reality, the node might be merely operating slowly. As we showed in our experiments, with reasonable heartbeat intervals, Saucr's false positive rate is low. However, we note that Saucr does not strictly require low false positive rates. In case the detector incorrectly predicts that a slow node has crashed, it has no effects on Saucr's correctness. Specifically, the system may switch to slow mode, or the follower may flush its data to its disk; these reactions may only reduce performance momentarily, but do not affect correctness guarantees in any way.

Further, Saucr differs from prior work on building failure detectors. While these detectors focus only on quickly detecting failures, Saucr also takes corrective steps (such as flushing data buffers to disk or switching to slow mode) in addition to performing timely failure detection. Fortunately, flushing dirty data in the background to the disk reduces the amount of data that needs to be written upon detection, reducing reaction time.

6.3.2 Situation-Aware Updates

The general idea of dynamically transitioning between different modes is common in real-time systems [43]. Similarly, the idea of fault-detection-triggered mode changes has been used in cyber-physical distributed systems [53]. However, we do not know of any previous work that dynamically adapts a distributed update protocol to the current situation.

Many practical systems statically define whether updates will be flushed to disk or not [21, 60, 76, 197]. This is a one-time configuration that needs to be set before starting the system; if one needs to change this setting, a restart of the entire system may be required. A few systems, such as MongoDB, provide options to specify the durability of a particular request [165]. For example, in MongoDB, the durability of a particular write request can be configured with different properties using the `writeConcern` and `journal` parameters. However, such dynamicity of whether the request will be persisted or buffered is purely client-driven: the storage system does not automatically make any such decisions, depending on the current situation of the system.

6.3.3 Situation-Aware Recovery

SAUCR's recovery is similar to recovery approaches proposed by prior systems. First, RAMCloud's recovery [176, 216] has a similar flavor to SAUCR. However, the masters in RAMCloud always construct their data from remote backups. In contrast, SAUCR performs mode-specific recovery: a SAUCR node may recover its data either from its local disk or from the remote replicas depending upon the mode in which it operated before it crashed.

SAUCR's recovery is also similar to the recovery of viewstamped replication [140]. However, SAUCR's recovery differs from that of viewstamped replication in two ways. First, in viewstamped replication, a recovering

node waits for a majority responses before it moves to the recovered state, while in Saucr, a recovering node has to wait only for a bare minority responses. Second, and more importantly, in viewstamped replication, a responding node can readily be in the recovered state only if it has not yet crashed. In contrast, in Saucr, a node can readily be in the recovered state in two ways: either it could have operated in fast mode and not failed yet, or it might have operated in slow mode previously or flushed to disk. These differences improve Saucr's availability. Saucr's recovery is also similar to how Ctrl recovers corrupted data from redundant copies [10].

6.3.4 Performance Optimizations in RSM systems

A prior body of work has optimized majority-based RSM systems by exploiting network properties [184]. This approach notes that if the network provides ordering guarantees, then the explicit ordering produced by the leader is extraneous and can be removed from the protocol. Thus, in the common case, the clients would send their requests to all the replicas directly, and the network would deliver (with high probability) these request in the same order across replicas. Such an optimization reduces the time to commit a request from two round trips to one round trip. If reordering arises, then the system rolls back to an older state. Similar approaches have tried to enforce such ordering primitives in the network [139]. Other optimizations to improve the performance on replication protocols that exploit commutativity [166] and parallelism [123] have also been proposed.

However, to the best of our knowledge, these systems are only memory-durable: they do not write any data to disk (sometimes not even in the background). Saucr can augment such systems; specifically, Saucr's fast mode can preserve the common-case performance properties of these systems and when failures arise, by switching to slow mode or flushing to disk can provide stronger guarantees.

A few systems [40, 59, 174] realize that synchronous disk writes are a major bottleneck in realizing highly performant RSM systems. These systems have proposed techniques (e.g., batching) that make disk I/O efficient. Saucr's implementation includes such optimizations in its slow mode.

6.4 Testing Distributed Systems

Our work on testing the effects of file-system crash behaviors in distributed systems under correlated crashes is related to several prior tools that aim to discover bugs in distributed systems. However, Pace is different from all previous tools through its focus on the interaction between the distributed system and the local file system.

Distributed Model Checkers. Researchers and practitioners alike have designed a variety of distributed model checkers [73, 102, 104, 135, 147, 237, 238] to find vulnerabilities in distributed systems. The main target of these model checkers is to test the various possible interleavings of events (such as messages between nodes). In this regard, the bugs found by these tools are similar to concurrency bugs, but in a distributed setting. Given that testing all possible reorderings is expensive, prior work on model checking has proposed ways to tackle the state-space explosion problem such as dynamic partial order reduction [237].

One deficiency in most model checkers is that they do not introduce node crashes and reboots as events that can occur during an execution; they only reorder messages. SAMC [135], in addition to reordering messages, also introduces node crashes and reboots as events that need to be permuted in a distributed execution. This further aggravates the state-space explosion problem. SAMC uses semantic information to prune the state space. In SAMC, such semantic information requires testers to write protocol-specific rules for a target system and then be given as input to

the model checker.

Testing Network Partitions. More recently, a set of new tools to test the effect of network partitions in distributed systems have gained attention. The most notable one among these tools is Jepsen [129]. Jepsen runs a workload on a target system (such as ZooKeeper) and then randomly injects network partitions. The framework then checks if guarantees (such as no loss of acknowledged data) are violated in the presence of partitions. More recent frameworks such as NEAT also inject network partition more systematically and can inject many different realistic ways in which network partitions manifest in real deployments [8].

Pace is complementary to existing tools: bugs that arise due to file-system crash behaviors discovered by Pace cannot be discovered by existing model checkers or partition-injection frameworks; similarly, bugs that arise due to network message re-orderings or partitions cannot be discovered by Pace. Previous tools focus solely on permuting different events in a distributed execution (such as network messages, node crashes, etc.) and thus cannot understand the interaction of distributed update and recovery protocols and local-storage protocols. To our knowledge, our work on Pace is the first to consider file-system behaviors in the context of distributed systems.

Similar to most prior tools, Pace uses semantic information about the distributed protocols to reduce the state-space explosion problem. However, in contrast to tools such as SAMC, that require testers to code the semantic information, Pace uses only high-level protocol-awareness and does not require such semantic information to be coded.

6.5 Crash Vulnerabilities in Single-node Applications

Our focus on file-system crash behaviors in Chapter 5 was motivated by recent research that demonstrated that file-system crash behaviors are largely undocumented and that they affect application correctness [9, 41, 181–183, 224]. The abstract-persistence-model (APM) specifications that we used in Pace were derived from our previous work [181].

Our work on finding crash vulnerabilities in distributed systems is related to previous efforts in discovering crash vulnerabilities in single-node settings. In our previous work (that is not a part of this dissertation), we developed Alice, a tool that can uncover crash vulnerabilities in single-machine applications. Using the tool, we identified that many single-machine applications, such as LevelDB, SQLite, BerkeleyDB, etc., are vulnerable to system crashes. Tools like Alice cannot be directly applied to distributed systems as they do not track cross node dependencies. If applied, such tools may report spurious vulnerabilities. Although such tools can be applied in stand-alone mode like in ZooKeeper [250], many code paths would not be exercised and thus miss critical vulnerabilities.

Similar to our analysis using Pace, Zheng et al. [244] find crash vulnerabilities in databases. However, unlike our work, Zheng et al. focus only on single-machine applications. Further, the methodology used in that work does not systematically explore all states that can occur in an execution because they do not model file-system behavior. Instead, their framework works atop already-implemented storage stacks and so finds vulnerabilities that can commonly occur. It is difficult for such tools to reproduce the vulnerabilities found by Pace. In contrast, Pace models the file system using an APM and thus can check how a distributed storage system will work on any current or future file system.

7 Conclusions and Future Work

In this chapter, we first summarize each part of this dissertation (§7.1). Then, we discuss the various lessons we learned through the course of this dissertation work (§7.2), present possible future work (§7.3), and finally conclude (§7.4).

7.1 Summary

This dissertation is comprised of three parts. In the first part, we showed how current approaches to handle storage fault do not work correctly, motivating the need for a new solution. We developed Par, a new, principled approach for distributed storage systems to recover from storage faults. In the second part, we studied the reliability and performance characteristics of existing replication approaches and showed that current approaches present unsavory tradeoffs. We developed Saucr, a new replication scheme that offers both strong reliability and high performance. Finally, we studied the effects of file-system crash behaviors in distributed systems and showed that many systems are vulnerable. We pointed to some possible ways to fix these problems. We now summarize each of these parts.

7.1.1 Storage Faults - Analysis and Solution

We first analyzed how distributed systems behave in the presence of storage faults, i.e., cases where portions of data persisted on the storage devices of the nodes could be inaccessible or corrupted on later accesses. We focused on RSM systems, an important class of distributed systems. Through fault-injection and qualitative analyses of practical systems and prior approaches, we developed the RSM recovery taxonomy. Our analyses revealed that none of the existing approaches are adequate to handle storage faults: they either lead to safety violations or unavailability. We found that the reason that currently employed approaches fall short is that they do not use any protocol-level knowledge to perform recovery.

To address this problem, we presented protocol-aware recovery (PAR), a new approach to handling storage faults in distributed systems. PAR exploits protocol-specific knowledge of the underlying distributed system to recover from storage faults correctly. We designed corruption-tolerant replication (CTRL), a protocol-aware recovery approach for RSM systems. We implemented CTRL in two systems (LogCabin and ZooKeeper) that are based on two different consensus protocols. Through rigorous experiments, we showed that CTRL correctly recovers from a range of storage faults, preserving safety and offering high availability. We also demonstrated that the reliability improvements of CTRL come with little to no performance overheads in the common case (up to 10% on HDDs and 4% on SSDs).

7.1.2 Crash Resiliency and Performance - Analysis and Solution

In the second part of this thesis, we analyzed the resiliency and performance characteristics of existing approaches to replication. We found that a dichotomy exists with respect to how and where current approaches

store system state. Disk-durable protocols replicate state to persistent storage on many nodes, while memory-durable approaches replicate data only to volatile memory. We found that this choice of where to commit data in a replicated system has significant implications for crash-resiliency and performance. Through measurements and analysis, we found that current approaches either provide strong crash resiliency or high performance, but not both. Specifically, disk-durable methods offer strong reliability but impose high overheads; in contrast, memory-durable approaches deliver high performance but suffer from data loss or unavailability in the presence of failures.

We presented situation-aware updates and crash recovery (SAUCR), a new solution to solve the unsavory tradeoff between crash resiliency and performance in replication protocols. Unlike existing replication approaches, SAUCR adapts the update protocol to the current situation: with many nodes up, SAUCR operates in a fast mode, buffering updates in memory; when failures arise, SAUCR switches to a slow mode, flushing updates to disk. Such reactiveness and situation-awareness enable SAUCR to achieve high performance similar to a memory-durable protocol while providing strong durability and availability guarantees similar to a disk-durable protocol. We implemented a prototype of SAUCR in ZooKeeper. Through rigorous testing, we demonstrated that SAUCR significantly improves durability and availability compared to memory-durable ZooKeeper. We also showed that SAUCR's reliability improvements come at little or no cost: SAUCR's overheads are within 0%-9% of memory-durable ZooKeeper. Compared to disk-durable ZooKeeper, we showed that, with a slight reduction in availability in rare cases, SAUCR improves performance by 25× to 100× on HDDs and 2.5× on SSDs.

7.1.3 File-system Crash Behaviors in Distributed Systems

In the final part of this thesis, we explored the effects of file-system crash behaviors in distributed systems. We examined these effects under a restricted correlated failure scenario where all replicas of a system crash together and recover later. To reason about the persistent states that can arise on the nodes during such a failure, we developed PACE.

PACE produces all possible correlated crash states that can occur in a distributed execution. It then introduces file-system crash behaviors using an abstract persistence model (APM). The relaxations from an APM can be applied in numerous ways on the nodes, and so one needs to test a vast state space. To address this problem, we develop a set of generic rules based upon how distributed update protocols work; with these rules, PACE reduces checking time from days to hours in some cases. We applied PACE to eight widely used systems and discovered 26 serious vulnerabilities such as data loss and unavailability in many of these systems. Many of these vulnerabilities have been acknowledged and fixed by developers. We identified the file-system properties that modern distributed storage system expect from the file system for their local update and recovery protocols to work correctly. Finally, we pointed to some ways in which these vulnerabilities can be fixed.

7.2 Lessons Learned

We now present a list of general lessons we learned while working on this dissertation.

7.2.1 The Importance of Measuring and then Building

In all three parts of this dissertation, we first analyzed the reliability and performance characteristics of real systems. Then, using the lessons learned through our study, we improved the resiliency of the studied systems in a principled way. In hindsight, we believe this two-step "*measure, then build*" process to conducting research was immensely useful.

We believe there are three major advantages to the two-step approach. First, it forced us to think about how to construct tools that can help us do the measurements and analysis. Generalizing the tools to make them work across many different systems leads to a new general methodology, which we believe is a useful research output. For example, the Pace tool we described in Chapter 5 was the result of our attempts to automate reasoning about correlated crashes. Similarly, our efforts to make Pace work well in practice across many systems led to the general idea of exploiting protocol knowledge for efficient exploration.

Second, the problems that we find across many systems by applying the tools, even without the solutions, can be interesting and useful to the community. For example, in our work on analyzing file-system crash behaviors, although we did not devise solutions to solve the problems we found, the vulnerabilities themselves led to many discussions and a few patches [82–84, 143, 200, 201]. Similarly, the safety violations that we discovered in Chapter 3 pointed to serious problems in well-tested systems.

Finally, the same measurement framework that we build to conduct our studies can help evaluate the efficacy of our solutions. For example, in our work on building Par, we used the same fault-injection framework that we used to study existing systems to evaluate the correctness of Ctrl.

Overall, we believe that first analyzing existing systems to understand the current state of affairs is key to find real, important problems to work on. Then, as the second step, insights from the study can guide how new solutions can be devised to solve the problem at hand.

7.2.2 Perspectives on Working on Widely Used Systems

We believe performing measurement- or study-driven work in real, widely used systems can be useful. In the first half of Chapters 3 and 4, and in Chapter 5, we studied widely used systems (such as ZooKeeper, Redis, MongoDB, etc.) In the second half of these chapters, we improved the resiliency of these systems through new ideas. Instead of implementing these solutions in a system built from scratch, we consciously opted to modify the real systems (in which we found the problems in the first place).

We believe there are many advantages to studying and modifying existing, widely used systems. The first and main advantage is that performing measurements on widely used systems (e.g., ZooKeeper) can give assurance that the problem at hand is important to solve. Given that these systems are widely used, even finding a handful of problems, and fixing them has the potential to improve reliability for many deployments.

Second, when implementing new solutions, one can get enough confidence that the idea is viable and can be implemented in a real system; it also confirms that the assumptions made during design are reasonable.

Next, we felt that developers of most systems that we examined were quite responsive to bug reports and discussions. We believe this is due to the large user base that is already active on discussion and bug-reporting forums. For example, developers of various systems were interested in knowing more about the vulnerabilities discovered by PACE and our storage-fault-injection framework [245–249].

Finally, in an existing, well-maintained system, basic infrastructure pieces (e.g., RPC mechanisms) are already implemented and heavily tested, helping us avoid spending too much time on developing infrastructure code. For instance, when implementing CTRL in LogCabin and ZooKeeper, we reused the existing RPC infrastructure to piggyback information about storage faults.

There are two minor drawbacks to working on an already existing system. First, one needs to understand a lot of code before making changes to the system; however, given the benefits, we believe this is a reasonable price to pay. Second, the solutions that we implement may carry the inherent problems in the current implementation of the system. For example, when we implemented SAUCR in ZooKeeper, the software overheads of existing code were too high when running on fast storage devices; this resulted as less-pronounced benefits of SAUCR atop SSDs. However, we believe that this is a small problem compared to the benefits; on the positive side, such problems point to opportunities for new optimizations.

7.2.3 The Importance of Paying Careful Attention to Failures

Traditionally, in distributed systems, two failure models at the two opposite ends of the spectrum have gained attention: fail-stop failures and Byzantine failures. While the fail-stop failure model is simple to understand and reason about, real-world systems seldom fail in such simplistic ways. At the other end of the spectrum, Byzantine failures include extreme failures such as adversarial behavior and security attacks. However, there exists many complex and realistic failure scenario that are prevalent in today's large-scale deployments; however, such failures are often overlooked: distributed systems are rarely tested against these failures. In this dissertation, we analyzed how systems react to realistic and often ignored failures such as storage faults and file-system crash behaviors.

We believe it is important to consider and pay careful attention to how failures arise in real deployments. We realized two benefits in this dissertation by doing so. First, by subjecting systems to realistic failures (such as storage faults), we discovered that even well-tested, widely used systems provide only a false sense of reliability. We believe the reason for this outcome is that the failures that we consider in this dissertation are sel-

dom tested by developers and practitioners. Second, by exploiting how failures actually arise, we designed new solutions. For example, SAUCR exploits the non-simultaneity of correlated failures to transition between its fast and slow modes, obtaining both high performance and strong guarantees.

Overall, we believe that realistic failure scenarios do not get the attention they deserve, and we believe this dissertation takes a first step towards addressing this problem. We believe that considering new failures models related to storage in distributed systems was largely possible because of our prior experience with problems that arise in the storage stack (for example, file-system crash behaviors and storage faults).

7.3 Future Work

In this section, we discuss directions in which work done in this dissertation can be extended.

7.3.1 Applying PAR to Other Systems

In Chapter 3, we applied the protocol-aware recovery approach to RSM systems. While this is an improvement in RSM systems to handle storage faults, we believe this is only a first step: other classes of distributed storage systems such as primary-backup (e.g., Redis), Dynamo-style-quorum systems (e.g., Cassandra), etc., are still vulnerable to storage faults.

In a recent work, Ganesan et al., analyze how many distributed storage systems react to storage faults [91, 92]. The study concludes that many systems do no effectively utilize the inherent redundancy in distributed systems to recover from storage faults. We believe the PAR approach can be applied to these systems to improve their improve their reliability. However, applying PAR to new classes of systems requires more work. For example, implementations of primary-backup protocols have subtle dif-

ferences across different systems, which makes it challenging to distinguish protocol-level attributes from system-specific details. In contrast, properties of RSM protocols have been well documented, making implementations mostly uniform; thus, identifying protocol attributes that the recovery approach must exploit was not a daunting challenge. However, despite these challenges, we believe it might be worthwhile to implement PAR and examine how the reliability of these systems can be improved.

7.3.2 Minimizing Software Overheads on Fast Storage

In Chapter 4, we designed and implemented SAUCR in ZooKeeper. Our experiments showed that SAUCR provides significant performance advantages compared to disk-durable ZooKeeper. On hard-disk drives, for a write-heavy workload, SAUCR was about 100× faster than disk-durable ZooKeeper. In contrast, the improvements were not so pronounced when running on fast flash-based SSDs: SAUCR was only about 2× faster. As we discussed, this problem is not specific to SAUCR's implementation; when running atop SSDs, even memory-durable ZooKeeper (which is less reliable than SAUCR) is only 2× faster than disk-durable ZooKeeper.

The reason for this is that the software overheads in ZooKeeper's current implementation become dominant atop fast storage devices. We believe the underlying cause for this effect is that ZooKeeper was designed in the era of hard disks, and thus the software inefficiencies were minuscule compared to the storage latency. However, with faster storage devices, software overheads contribute to much of the latency. We believe this trend is highly likely to continue given the advent of even faster devices such as Intel Optane SSDs and NVM devices. Thus, we believe these software overheads in RSM implementations must be reconsidered in the era of faster storage devices similar to how researchers have revisited the performance of LSM trees on SSDs [133].

Overall, we believe there is an opportunity for systems researchers to

realize a new RSM implementation that is performant atop faster storage technologies. We believe efficient concurrency control (e.g., fine-grained locks instead of coarse-grained locks), minimal dependency between request-processing stages, and few serialization points (for example, to establish the request ordering) are key to such an efficient implementation.

7.3.3 Reasoning about Partial Failures

In Chapter 5, we analyzed how file-system crash behaviors affect distributed systems under correlated crash scenarios where all replicas crash and recover together. While this is an important failure scenario to study, we believe it might be interesting to explore how the persistent states on one node affect the entire system when that machine fails while the rest of the system continues to operate.

However, examining this failure scenario using a tool such as Pace (in its current form) is challenging. In Pace, we run a workload once and record the traces at the replicas. Then, we find the correlated crash states that could occur in this execution by calculating the valid cross product of the possible crash states at the individual replicas. However, in the new failure model, a part of the system would run while another part crashes; with this, the one-time record-and-replay methodology would not work. Specifically, when a node crashes at some point, the other nodes might react in a certain way (possibly sending new messages, and writing new data to their disks) and when the node crashes at a different point, then the other nodes might react in a different way. Thus, an active way of injecting crashes must be adopted. Another challenge is deciding when exactly to crash a node; one obvious choice is when the node's persistent state changes due to a file-system operation such as a `write`; in addition, the tool could crash the node at random points, helping one to find timing-related bugs in addition to bugs related to persistent state.

7.3.4 Crash Consistency in Distributed Transactions

In Chapter 5, we studied the crash consistency of many practical distributed systems. However, we focused only on problems within a single shard of data. We believe studying crash consistency across many shards can be valuable. Specifically, we believe it might be worthwhile to study if distributed transactions, which must update data on many shards correctly to successfully commit transactions, work correctly in the presence of crashes. Distributed transactions form the core of many critical services in the cloud and thus testing their reliability can reveal many problems that have eluded the attention of practitioners. While transactions and crash recovery have been well studied in the database community [149, 157, 158], we believe testing modern systems that depend upon local file systems to commit transactions across shards is a valuable exercise.

7.3.5 Studying Interdependency Faults

Complex distributed systems are not always built from scratch; instead, developers prefer to utilize pre-existing building blocks and layer them to build the target system. Such an approach facilitates developer velocity and helps avoid reimplementation of same functionality [109].

The Frangipani distributed file system [225], built atop the Petal distributed virtual disks [134], is a great example of such a composite system. In our work in this dissertation, we noted that several modern distributed systems are also built this way: by combining two or more building blocks. For example, Kafka depends upon ZooKeeper for storing its metadata such as broker information, client sessions, etc., and delegates problems such as leader election to ZooKeeper [23]. Similarly, BigTable depends on GFS for storage, the HBase data store is layered atop HDFS, and HDFS depends on ZooKeeper for automatic failover of its namen-

ode [22]. Sometimes, as high as few tens of independent services are composed together to provide a single high-level service (e.g., microservices at Netflix [152]).

Although such layered architectures ease the construction of complex systems, they come at a cost: a complex interaction between the building blocks. We believe an emerging class of faults that we call *interservice dependency faults* must be studied to understand how such composition works in the presence of failures. These faults arise at the interaction points of building blocks that constitute a composite distributed system. For example, it might be worthwhile to explore cases where ZooKeeper does not behave in a certain way (e.g., because of failures) that Kafka expects it to. A few pieces of anecdotal evidence already exist where interservice dependency faults have caused catastrophic outcomes in distributed systems. For example, at Google, an update to one Service, say A (unrelated to another service B) started creating more socket connections to service A, which exhausted the file descriptors in jobs of service A [151]. Similar cases have been documented in other systems too [2–4]. Given this, we believe it is imperative to study this important class of failures in distributed systems.

7.4 Closing Words

Many applications and users are entrusting distributed storage systems with their critical data. Despite the importance of these systems, only scant attention has been paid to problems that arise at their storage interface. This dissertation takes a step towards understanding and addressing the reliability and performance challenges at the interaction points between distributed systems and the storage stack.

Through our studies, we showed that even widely used systems provide only a false sense of reliability; in the presence of realistic failures,

these systems can lose or corrupt data, or become unavailable. Most of the problems we find suggest that even the obvious expectations one has about distributed systems, such as redundancy improves reliability, are surprisingly hard to realize correctly in practice. We believe validating such commonly held assumptions through careful measurements and analysis of existing systems is a crucial first step before building new solutions. Using the insights from our studies, and by paying careful attention to how failures arise in deployments, we showed how new solutions can be applied to distributed systems to improve their resiliency to problems at the storage interface without impacting performance.

Although our dissertation takes an important step towards understanding the interaction of distributed systems and the storage layer, it is only a first step. The storage landscape is changing at a tremendous pace, with many new storage technologies such as the 3D XPoint memory [1, 87, 105] and QLC NAND flash [155] becoming available commercially. These new technologies will inevitably find their way into data centers, presenting a new set of challenges for distributed storage systems. We believe the ideas presented in this thesis can help solve some of these future challenges.

Bibliography

[1] 3D XPoint. https://en.wikipedia.org/wiki/3D_XPoint.

[2] Kafka getting stuck creating ephemeral node it has already created when two zookeeper sessions are established in a very short period of time. https://issues.apache.org/jira/browse/KAFKA-1387.

[3] Kafka server can miss zookeeper watches during long callbacks. https://issues.apache.org/jira/browse/KAFKA-1155.

[4] Kafka unable to reconnect to zookeeper behind an ELB. https://issues.apache.org/jira/browse/ZOOKEEPER-2184.

[5] Necessary step(s) to synchronize filename operations on disk. http://austingroupbugs.net/view.php?id=672.

[6] Pace Tool and Results. http://research.cs.wisc.edu/adsl/Software/pace/.

[7] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. *Distributed Computing*, 18(5):387–408, 2006.

[8] Ahmed Alquraan and Hatem Takruri and Mohammed Alfatafta and Samer Al-Kiswany. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Conference*

on Operating Systems Design and Implementation (OSDI '18), Carlsbad, CA, October 2018.

[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, pages 20–20, Berkeley, CA, USA, 2015. USENIX Association.

[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 C. Arpaci-Dusseau, and Remzi H. 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] Amazon Elastic Compute Cloud. Regions and Availability Zones.

https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

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

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

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

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

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

[19] Apache Accumulo Users. Setting ZooKeeper forceSync=no. http://apache-accumulo.1065345.n5.nabble.com/setting-zookeeper-forceSync-no-td7758.html.

[20] Apache Cassandra. Cassandra Replication. http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html.

[21] Apache Cassandra. Cassandra Wiki: Durability. https://wiki.apache.org/cassandra/Durability.

[22] Apache Hadoop. HDFS High Availability Using the Quorum Journal Manager. https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html.

[23] Apache Kafka. kafka Documentation - ZooKeeper. https://kafka.apache.org/documentation/#zk.

[24] Apache ZooKeeper. Applications and Organizations using ZooKeeper. https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy.

[25] Apache ZooKeeper. QuorumPacket Class. http://people.apache.org/~larsgeorge/zookeeper-1075002/build/docs/dev-api/org/apache/zookeeper/server/quorum/QuorumPacket.html.

[26] Apache ZooKeeper. ZooKeeper Overview. https://zookeeper.apache.org/doc/r3.5.0-alpha/zookeeperOver.html.

[27] ArchLinux. f2fs. https://wiki.archlinux.org/index.php/F2FS.

[28] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

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

[30] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems (2nd Edition)*, pages 55–96. ACM Press/Addison-Wesley Publishing Co., 1993.

[31] 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.

[32] 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.

[33] 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.

[34] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, CA, June 2009.

[35] Lakshmi Narayanan Bairavasundaram. *Characteristics, Impact, and Tolerance of Partial Disk Failures*. PhD thesis, University of Wisconsin, Madison, 2008.

[36] Mehmet Bakkaloglu, Jay J Wylie, Chenxi Wang, and Gregory R Ganger. On Correlated Failures in Survivable Storage Systems. Technical Report CMU-CS-02-129, School of computer science, Carnegie-Mellon University, 2002.

[37] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[38] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[39] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: An Exercise in Distributed Computing. *Commun. ACM*, 25(4):260–274, April 1982.

[40] 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.

[41] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, Atlanta, GA, April 2016.

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

[43] Alan Burns. System Mode Changes - General and Criticality-Based. In *Proc. of 2nd Workshop on Mixed Criticality Systems (WMC)*, pages 3–8, 2014.

[44] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[45] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA-15)*, San Francisco, CA, February 2015.

[46] Marco Canini, Vojin Jovanovic, Daniele Venzano, Gautam Kumar, Dejan Novakovic, and Dejan Kostic. Checking for Insidious Faults in Deployed Federated and Heterogeneous Distributed Systems. Technical report, 2011.

[47] Cassandra. Apache Cassandra. `https://academy.datastax.com/resources/brief-introduction-apache-cassandra`.

[48] 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.

[49] Ceph. New in Luminous: BlueStore. `https://ceph.com/community/new-luminous-bluestore/`.

[50] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, August 2007.

[51] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[52] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes,

and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. pages 15–15, 2006.

[53] Ang Chen, Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan. Fault Tolerance and the Five-second Rule. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*, Kartause Ittingen, Switzerland, May 2015.

[54] Vijay Chidambaram. *Orderless and Eventually Durable File Systems*. PhD thesis, University of Wisconsin, Madison, 2015.

[55] 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)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[56] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[57] Chris Mellor. Storage with the speed of memory? XPoint, XPoint, that's our plan. `https://www.theregister.co.uk/2016/04/21/storage_approaches_memory_speed_with_xpoint_and_storageclass_memory/`.

[58] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX '13)*, San Jose, CA, June 2013.

[59] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright Cluster

Services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[60] CockroachDB. CockroachDB Cluster Settings. `https://www.cockroachlabs.com/docs/stable/cluster-settings.html`.

[61] ComputerWorldUK. Lightning strikes Amazon and Microsoft data centres. `http://www.computerworlduk.com/galleries/infrastructure/ten-datacentre-disasters-that-brought-firms-offline-3593580/#5`.

[62] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[63] 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.

[64] CoreOS. etcd Guarantees. `https://coreos.com/blog/etcd-v230.html`.

[65] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical hardening of crash-tolerant systems. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 453–466, Boston, MA, 2012. USENIX.

[66] Flaviu Cristian, Houtan Aghili, Raymond Strong, and Danny Dolev. *Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement*. Citeseer, 1986.

[67] DataCenterDynamics. Lessons from the Singapore Exchange failure. http://www.datacenterdynamics.com/power-cooling/lessons-from-the-singapore-exchange-failure/94438.fullarticle.

[68] DataCenterKnowledge. Lightning Disrupts Google Cloud Services. http://www.datacenterknowledge.com/archives/2015/08/19/lightning-strikes-google-data-center-disrupts-cloud-services/.

[69] Datastax. Netflix Cassandra use case. http://www.datastax.com/resources/casestudies/netflix.

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

[71] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[72] 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. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[73] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 249–262, Santa Clara, CA, Feb 2016. USENIX Association.

[74] Diego Ongaro. Raft TLA+ Specification. https://github.com/ongardie/raft.tla.

[75] Docker. Docker. https://www.docker.com/.

[76] Elasticsearch. Translog Settings. https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-translog.html#_translog_settings.

[77] epaxos. epaxos source code. https://github.com/efficient/epaxos.

[78] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[79] etcd. etcd. https://coreos.com/etcd.

[80] Etcd. Etcd: Production users. https://coreos.com/etcd/docs/latest/production-users.html.

[81] Etcd. Etcd Tuning. https://coreos.com/etcd/docs/latest/tuning.html.

[82] etcd. Possible cluster unavailability on few file systems. https://github.com/coreos/etcd/issues/6379.

[83] etcd. Possible cluster unavailbility. https://github.com/coreos/etcd/issues/6378.

[84] etcd. Possible data loss – fsync parent directories. https://github.com/coreos/etcd/issues/6378.

[85] 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.

[86] Flavio Junqueira. [ZooKeeper-user] forceSync=no. http://grokbase.com/p/zookeeper/user/126g0063x4/forcesync-no.

[87] Forbes. Intel And Micron Announce Breakthrough Faster-ThanFlash 3D XPoint Storage Technology. https://www.forbes.com/sites/antonyleather/2015/07/28/intel-and-micron-announce-breakthrough-faster-then-flash-3d-xpoint-storage-technology/#60dd71c63ec2.

[88] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[89] 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 Conference on File and Storage Technologies (FAST 14)*, pages 295–308, Santa Clara, CA, 2014. USENIX.

[90] 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 Conference on File and Storage Technologies*, FAST'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[91] 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 File-System Faults. *ACM Trans. Storage*, 13(3):20:1–20:33, September 2017.

[92] 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.

[93] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 21–21, Berkeley, CA, USA, 2007. USENIX Association.

[94] 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.

[95] Google. Google Cloud Status. `https://status.cloud.google.com/incident/compute/15056#5719570367119360`.

[96] Google. Google Cluster Data. `https://github.com/google/cluster-data`.

[97] Google. LevelDB. `https://code.google.com/p/leveldb/`.

[98] Google Code University. Introduction to Distributed System Design. `http://www.hpcs.cs.tsukuba.ac.jp/~tatebe/lecture/h23/dsys/dsd-tutorial.html`.

[99] Matthias Grawinkel, Thorsten Schafer, Andre Brinkmann, Jens Hagemeyer, and Mario Porrmann. Evaluation of Applied Intra-disk Redundancy Schemes to Improve Single Disk Reliability. In *Proceedings of the 19th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Washington, DC, July 2011.

[100] Kevin M Greenan, Darrell DE Long, Ethan L Miller, Thomas Schwarz, and Avani Wildani. Building Flexible, Fault-Tolerant Flash-Based Storage Systems. In *The 5th Workshop on Hot Topics in System Dependability (HotDep '09)*, Lisbon, Portugal, June 2009.

[101] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*, New York, New York, December 2009.

[102] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 225–238, Berkeley, CA, USA, 2011. USENIX Association.

[103] Haryadi Gunawi. *Towards Reliable Storage Systems*. PhD thesis, University of Wisconsin, Madison, 2009.

[104] 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 Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM.

[105] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[106] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, May 2005.

[107] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association.

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

[109] Tyler Harter. *Emergent Properties in Modular Storage: a Study of Apple Desktop Applications, Facebook Messages, and Docker Containers*. PhD thesis, University of Wisconsin, Madison, 2016.

[110] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[111] Henry Robinson. Consensus Protocols: Paxos. `http://the-paper-trail.org/blog/consensus-protocols-paxos/`.

[112] 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.

[113] iNexus. iNexus. https://github.com/baidu/ins.

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

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

[116] John Goerzen. Silent Data Corruption Is Real. http://changelog.complete.org/archives/9769-silent-data-corruption-is-real.

[117] Jonathan Corbet. Responding to ext4 journal corruption. https://lwn.net/Articles/284037/.

[118] 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.

[119] 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.

[120] Kafka. Kafka Disks and Filesystem. https://kafka.apache.org/081/ops.html.

[121] Kafka. Possible data loss. https://issues.apache.org/jira/browse/KAFKA-4127.

[122] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Non-volatile Memory with NoveLSM. In *Proceedings of the USENIX Annual Technical Conference (USENIX '18)*, Boston, MA, July 2018.

[123] 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.

[124] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, CA, April 2004.

[125] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, CA, April 2004.

[126] Ken Birman. What we can learn about specifications from ZooKeeper's asynchronous mode, and its unsafe ForceSync=no option? `http://thinkingaboutdistributedsystems.blogspot.com/2017/09/what-we-can-learn-from-zookeepers.html`.

[127] Madhukar Korupolu and Rajmohan Rajaraman. Robust and Probabilistic Failure-Aware Placement. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 213–224. ACM, 2016.

[128] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. HAFT: Hardware-assisted Fault Tolerance. In

Proceedings of the EuroSys Conference (EuroSys '16), London, United Kingdom, April 2016.

[129] Kyle Kingsbury. Jepsen. `http://jepsen.io/`.

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

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

[132] Butler Lampson and Howard Sturgis. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California, 1979.

[133] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[134] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, MA, October 1996.

[135] 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)*, pages 399–414, Broomfield, CO, October 2014.

[136] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving Availability in Distributed Systems with Failure Informers. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, April 2013.

[137] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming Uncertainty in Distributed Systems with Help from the Network. In *Proceedings of the EuroSys Conference (EuroSys '15)*, Bordeaux, France, April 2015.

[138] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[139] 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.

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

[141] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND Flash-based SSDs via Retention Relaxation. February 2012.

[142] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 485–500. USENIX Association, 2016.

[143] LogCabin. Cluster unavailable due to power failures. `https://github.com/logcabin/logcabin/issues/221`.

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

[145] Jacob R Lorch, Atul Adya, William J Bolosky, Ronnie Chaiken, John R Douceur, and Jon Howell. The SMART Way to Migrate Replicated Stateful Services. In *Proceedings of the EuroSys Conference (EuroSys '06)*, Leuven, Belgium, April 2006.

[146] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[147] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the EuroSys Conference (EuroSys '19)*, Dresden, Germany, March 2019.

[148] Parisa Jalili Marandi, Christos Gkantsidis, Flavio Junqueira, and Dushyanth Narayanan. Filo: Consolidated Consensus As a Cloud Service. In *Proceedings of the USENIX Annual Technical Conference (USENIX '16)*, Denver, CO, June 2016.

[149] Cris Pedregal Martin and Krithi Ramamritham. Toward Formalizing Recovery of (Advanced) Transactions. In *Advanced Transaction Models and Architectures*, pages 213–234. Springer, 1997.

[150] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New ext4 Filesystem:

Current Status and Future Plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[151] Matt Welsh. What I wish systems researchers would work on. http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html.

[152] Tony Mauro. Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[153] Marshall Kirk McKusick and Jeffery Roberson. Journaled Soft-updates. *Proceedings of EuroBSDCon*, 2010.

[154] 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.

[155] Micron. QLC NAND Technology. https://www.micron.com/products/advanced-solutions/qlc-nand.

[156] Microsoft Azure. Azure Availability Sets. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/tutorial-availability-sets#availability-set-overview.

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

[158] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[159] MongoDB. Introduction to MongoDB. https://docs.mongodb.org/manual/introduction/.

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

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

[162] MongoDB. MongoDB Platform Specific Considerations. https://docs.mongodb.org/manual/administration/production-notes/#platform-specific-considerations.

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

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

[165] MongoDB. MongoDB Write Concern. https://docs.mongodb.com/manual/reference/write-concern/.

[166] 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.

[167] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[168] 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.

[169] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 17–17, Berkeley, CA, USA, 2006. USENIX Association.

[170] 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.

[171] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[172] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

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

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

[175] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*, pages 305–320, Philadelphia, PA, 2014.

[176] 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.

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

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

[179] Thanumalayan Sankaranarayana Pillai. *Application Crash Consistency*. PhD thesis, University of Wisconsin, Madison, 2016.

[180] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017.

[181] 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 Sys-*

tems Design and Implementation (OSDI '14), Broomfield, CO, October 2014.

[182] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency. *Communications of the ACM*, 58(10):46–51, October 2015.

[183] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency: Rethinking the Fundamental Abstractions of the File System. *Communications of the ACM*, 13(7), July 2015.

[184] 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, May 2015.

[185] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, Yokohama, Japan, June 2005.

[186] 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.

[187] Redis. Instagram Architecture. `http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html`.

[188] Redis. Introduction to Redis. http://redis.io/topics/introduction.

[189] Redis. Redis. http://redis.io/.

[190] Redis. Redis at Flickr. http://code.flickr.net/2014/07/31/redis-sentinel-at-flickr/.

[191] Redis. Redis at GitHub. http://nosql.mypopescu.com/post/1164218362/redis-at-github.

[192] Redis. Redis at Pinterest. http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html.

[193] Redis. Redis Replication. http://redis.io/topics/replication.

[194] Redis. Virtual Memory – Redis . http://redis.io/topics/virtual-memory.

[195] Redis. Who's using Redis? http://redis.io/topics/whos-using-redis.

[196] RethinkDB. RethinkDB. https://www.rethinkdb.com/.

[197] RethinkDB. RethinkDB Settings - Durability. https://rethinkdb.com/docs/consistency/.

[198] Robert Harris. Data corruption is worse than you know. http://www.zdnet.com/article/data-corruption-is-worse-than-you-know/.

[199] RocksDB. RocksDB. http://rocksdb.org/blog/1967/integrating-rocksdb-with-mongodb-2/.

[200] Mongo RocksDB. Data loss – fsync parent directory on file creation and rename. https://github.com/mongodb-partners/mongo-rocks/issues/35.

[201] Mongo RocksDB. Mongodb - rocksdb data loss bug. https://groups.google.com/forum/#!topic/mongodb-dev/X9LQOorieas.

[202] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[203] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[204] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[205] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[206] 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.

[207] 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.

[208] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: The Growth of a Distributed System. *ACM Trans. Comput. Syst.*, 2(1):3–23, February 1984.

[209] Thomas Schwarz, Ahmed Amer, Thomas Kroeger, Ethan L. Miller, Darrell D. E. Long, and Jehan-FranÃ§ois PÃ¢ris. RESAR: Reliable Storage at Exabyte Scale. In *Proceedings of the 24th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, London, United Kingdom, September 2016.

[210] Romain Slootmaekers and Nicolas Trangez. Arakoon: A Distributed Consistent Key-Value Store. In *SIGPLAN OCaml Users and Developers Workshop*, volume 62, 2012.

[211] David A Solomon and Helen Custer. *Inside Windows NT*, volume 2. Microsoft Press Redmond, 1998.

[212] SQLite. SQLite Transactional SQL Database Engine. `http://www.sqlite.org/`.

[213] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, 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.

[214] Stackoverflow. Can ext4 detect corrupted file contents? `http://stackoverflow.com/questions/31345097/can-ext4-detect-corrupted-file-contents`.

[215] Stackoverflow. ZooKeeper Clear State. `http://stackoverflow.com/questions/17038957/org-apache-hadoop-hbase-pleaseholdexception-master-is-initializing`.

[216] Ryan Scott Stutsman. *Durability and Crash Recovery in Distributed In-Memory Storage Systems*. PhD thesis, Stanford University, 2013.

[217] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, CA, February 2010.

[218] 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.

[219] Amy Tai. *Leveraging Distributed Storage Redundancy in Datacenters*. PhD thesis, Princeton University, 2019.

[220] 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.

[221] 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.

[222] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 172–183, 1995.

[223] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, CA, February 2013.

[224] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Joo-young Hwang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-Level Consistency. In *The 9th Workshop on Hot Topics in System Dependability (HotDep '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[225] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, October 1997.

[226] Theodore Ts'o. What to do when the journal checksum is incorrect. `https://lwn.net/Articles/284038/`.

[227] TheRegister. Admin downs entire Joyent data center. `http://www.theregister.co.uk/2014/05/28/joyent_cloud_down/`.

[228] Eno Thereska, Hitesh Ballani, Greg OâĽ™Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: a Software-Defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[229] Twitter. Twitter Blogs. `https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time`.

[230] UWSysLab. VR Implementation. `https://github.com/UWSysLab/NOPaxos/tree/master/vr`.

[231] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2015.

[232] 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.

[233] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective Failure Analysis: Avoiding Correlated Failures in Peer-to-Peer Systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 362–367. IEEE, 2002.

[234] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[235] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-olerant Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[236] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and Preventing Inconsistencies in Deployed

Distributed Systems. *ACM Trans. Comput. Syst.*, 28(1):2:1–2:49, August 2010.

[237] Maysam Yabandeh and Dejan Kostić. DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems. Technical report, School of Computer and Communication Sciences, EPFL, 2009.

[238] 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 USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[239] YCombinator. Joyent us-east-1 rebooted due to operator error. `https://news.ycombinator.com/item?id=7806972`.

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

[241] Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Zettabyte Reliability with Flexible End-to-end Data Integrity. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST '13)*, Long Beach, CA, May 2013.

[242] 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.

[243] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, CA, February 2013.

[244] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 449–464, Berkeley, CA, USA, 2014. USENIX Association.

[245] ZooKeeper. Cluster unavailable on space and write errors. `https://issues.apache.org/jira/browse/ZOOKEEPER-2495`.

[246] ZooKeeper. Crash on detecting a corruption. `http://mail-archives.apache.org/mod_mbox/zookeeper-dev/201701.mbox/browser`.

[247] ZooKeeper. Possible Cluster Unavailability. `https://issues.apache.org/jira/browse/ZOOKEEPER-2560`.

[248] ZooKeeper. ZooKeeper cluster can become unavailable due to power failures. `https://issues.apache.org/jira/browse/ZOOKEEPER-2528`.

[249] ZooKeeper. Zookeeper service becomes unavailable when leader fails to write transaction log. `https://issues.apache.org/jira/browse/ZOOKEEPER-2247`.

[250] ZooKeeper. ZooKeeper Standalone Operation. `https://zookeeper.apache.org/doc/r3.3.3/zookeeperStarted.html#sc_InstallingSingleMode`.

[251] ZooKeeper Jira Issues. Unable to load database on disk when restarting after node freeze. https://issues.apache.org/jira/browse/ZOOKEEPER-1546.

[252] Zookeeper User Mailing List. Unavailability Issues due to Setting forceSync=no in ZooKeeper. http://zookeeper-user.578899.n2.nabble.com/forceSync-no-td7577568.html.

A
Proof of Impossibility of Last-Entry Disentanglement

In any log-based storage system, if the last entry in the log is corrupted, it is impossible to determine whether the corruption was due to a crash or a disk corruption (as we discussed in §3.2.4). We now present a proof of this claim; we first define the various elements necessary for the proof and then present the proof.

Log. We model the log L as two disjoint lists, one list L_e that stores entries and one list L_{id} that stores identifiers.

Identifiers. The identifier of a log entry contains vital information about that entry; this information helps CTRL's distributed protocol to recover corrupted entries from copies on other nodes.

Operations. Two kinds of operations update the log:

– *write*(v), which updates L_e or L_{id} (depending on if v is an entry or identifier).

– *fsync*() flushes all previous writes to disk.

Sequences. A *disentangled sequence* of transactions $\sigma = t_1, \ldots, t_n$, where $n > 1$ is one where each t_i is a subsequence of three operations: a_i^1, a_i^2, a_i^3, where:

- a_i^1 is of the form $\mathit{write}(e_i)$.
- a_i^2 is of the form $\mathit{write}(\mathit{id}_i)$.
- a_i^3 is of the form $\mathit{fsync}()$.

where e_i is the entry to be written and id_i is its respective identifier. For simplicity, we assume a single log.

Log appends. Suppose we are given a disentangled sequence $\sigma = t_1, \ldots, t_n$. We use L^I to denote the initial state of the log. We use σL^I to denote the state of the log after executing the sequence σ beginning from state L^I.

Corruption and Crash. We distinguish two *bad* events: corruptions *co* and crashes *cr*.

- A corruption co_i changes element e_i in L_e to some new e_i' where $e_i' \neq e_i$.
- We assume identifiers (id_i) cannot be affected by a corruption.
- We assume the identifiers can be atomically written to the disk because an identifier is much smaller than a single sector (i.e., $\mathit{write}(\mathit{id}_i)$ is atomic).
- We assume a crash cr_i can only happen between a_i^2 and a_i^3, i.e., right before the *fsync*, for a sequence $t_1, \ldots, t_n$, as defined above.

Given sequence σ, we use σ_{cr_i} to denote σ with a crash in t_i. If the system crashes during t_i, then no entries t_j would appear in the log for any $j > i$. Given σ, we use σ_{co_i} to denote σ with a corruption event co_i appended at the end.

Theorem A.1 (Disentanglement)**.** *Suppose we are given the disentangled sequence* σ *and log* L*.*

- ***Case 1****: Let* $L^1 = \sigma_{cr_n} L^I$*, and let* $L^2 = \sigma_{co_n} L^I$*. Suppose we are provided* L^I*,* σ*, and one of the logs* L^1 *and* L^2*. We cannot detect whether* σ_{cr_n} *or* σ_{co_n} *is the one that executed resulting in* L^1 *or* L^2*.*

- ***Case 2****: Let* $L^{co_i} = \sigma_{co_i} L^I$*, where* $i \in [1, n)$*. Provided* L^I, σ*, and* L^{co_i}*, we can conclude that* σ_{cr_j} *did not execute, where* $j \in [1, n]$*.*

Proof. First, we note that by being *able to detect* whether a crash or corruption happened, we mean that there exists a deterministic algorithm that will return whether a crash or corruption happened.

Case 1: We prove the first case with a simple construction. Let $\sigma = t_1$, where

$$t_1 = \mathit{write}(e_1), \mathit{write}(\mathit{id}_1), \mathit{fsync}()$$

Let L^I be the empty log. Let $L^1 = \sigma_{cr_1} L^I$ and $L^2 = \sigma_{co_1} L^I$.

Assume that when the crash cr_1 happened, only a strict subset of e_1 was written in addition to id_1. Let the strict subset of e_1 that was written be e_1'. The above condition can arise because $\mathit{write}(e_1)$ need not be atomic and writes can be reordered by the underlying file system on a crash. Now, assume that the corruption co_1 turns e_1 to e_1'.

We can now prove the first case by contradiction: Suppose there is an algorithm M that can take (*i*) the initial state of the log, (*ii*) the current state of the log, and (*iii*) the sequence of transactions σ that lead to the current state (minus *co* and *cr* events), and deterministically returns whether a crash or corruption happened. In the above example, $L^1 = L^2$ by construction. So, $M(L^I, L^1, \sigma) = M(L^I, L^2, \sigma)$. Therefore, no such M exists.

Case 2: Fix i, j as in theorem statement. Let $L^{cr_j} = \sigma_{cr_j} L^I$. Assume $L^{cr_j} = L^{co_i}$. If $j \neq i$, then entry e_i cannot be affected by the crash, and therefore the $L^{cr_j} \neq L^{co_i}$ If $j = i$, since $i < n$, then e_i is fixed by recovery. Therefore, $L^{cr_j} \neq L^{co_i}$.

□