TOWARDS APPLICATION RECOVERABILITY ATOP CLOUD-NATIVE STORAGE

BY

WENQING LUO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Adviser:

Assistant Professor Ramnatthan Alagappan
Assistant Professor Aishwarya Ganesan
Assistant Professor Tianyin Xu

ABSTRACT

Cloud-native applications, designed specifically for cloud environments, rely heavily on disaggregated storage services to manage their persistent state. These storage services enable applications to recover their state after failures, ensuring data durability and application availability. However, this thesis poses a critical question: *can cloud-native applications consistently and correctly recover after experiencing failures?* To investigate this, we have conducted preliminary research focusing on applications built on top of disaggregated file services.

Our main finding reveals that storage services demonstrate various *post-failure behaviors*, leading to unexpected states after application failures. These unpredictable states can cause severe consequences, such as data loss and application unavailability. We have identified that subtle interactions between the application and the storage service can significantly impact an application's recoverability.

Given the importance of recoverability in cloud-native applications, we aim to bring attention to this issue and outline the steps and vision to address it. In this thesis, we delve into understanding the interactions between applications and storage services, identifying post-failure behaviors exhibited by storage services, and uncovering application vulnerabilities based on the post-failure behaviors examined. Additionally, we developed an automated tool to emulate post-failure behaviors in application persistence protocols and detect recoverability issues.

Upon examining four applications atop two disaggregated file services, we identified vulnerabilities in all applications, resulting from the post-failure behaviors of the storage services. Our automated tool successfully reproduces all discovered vulnerabilities, and we aim to enhance the tool and apply it to more applications to uncover further vulnerabilities.

Ultimately, our research seeks to contribute to the growing body of knowledge on cloud-native applications and their recoverability, ensuring the development of more reliable, resilient, and fault-tolerant systems within the cloud.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my deep gratitude to my advisor, Assistant Professor Tianyin Xu. I first met Tianyin during my undergraduate studies, at a time when I held the stereotype that much of the research conducted in academia might not be strongly connected to the needs of industry. Tianyin's exceptional understanding of real-world production systems, gained through his industry experience, and his commitment to addressing real-world challenges, completely changed my perspective. Through each project we have collaborated on, I have been consistently convinced of the practicality and real-world significance of our work. To me, Tianyin did a great job on his professor role, which by his saying, on helping every student to be successful. On research, Tianyin always gave me the freedom to choose the research topics I wanted to work on and pushed me to think critically about the problems at hand. He has also been instrumental in building strong collaborative teams, connecting me with experts in the field. In addition to his research guidance, Tianyin has provided invaluable advice on life decisions. He never pressured me into making any particular decision but instead encouraged me to fully utilize my abilities and pursue what truly inspires me. When I was struggling with a decision, Tianyin helped me connect with different people, and through that, I happen to find the most suitable opportunity for my next journey. I feel incredibly fortunate and grateful to have had Tianyin as my advisor, and I will always cherish the experience of working with him.

I would also like to extend my thanks to my co-advisors, Assistant Professor Ramnatthan Alagappan and Assistant Professor Aishwarya Ganesan. I was fortunate enough to collaborate with Ram and Aishwarya before they joined the department, and I am delighted that they later chose to become part of the faculty and serve as my co-advisors. Their deep knowledge in system areas and critical thinking skills have been a constant source of inspiration to me. They always asked probing questions that helped me think about the problems in a more comprehensive manner. I truly enjoyed working with them to solve challenging problems, and without their guidance, I could not have completed this thesis.

Furthermore, I would like to express my gratitude to Lalith Suresh, who has offered valuable insights and assistance throughout multiple research projects we've worked together.

Finally, I would like to thank my fellow students, including Xudong Sun, Tyler Gu, Lilia Tang, Henry Zhu, Shuai Wang and so on. Exploring life at Urbana-Champaign with them and discussing research topics in the lab was a wonderful experience. I hope that everyone has a bright future and that we will have the opportunity to meet again in the future.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Applications are increasingly built in a cloud-native manner [1, 2, 3, 4, 5, 6], where they rely on services available in the cloud instead of implementing them on their own. One such critical service is persistent storage. Cloud-native applications depend upon disaggregated storage services for managing persistent state. The application runs on a compute cluster (within stateless VMs or containers) and persists state on a storage service disaggregated from compute.

The storage service ensures that the data is durable and consistent (applications observe only meaningful states) [6, 7]. Applications can thus seamlessly recover their state from the storage service after a failure and continue to operate.

Constructing applications in the above manner has become popular in the modern data center for two reasons. First, it eases development by offloading the hard concern of durability and consistency from applications. Applications need not implement complex storage or distributed protocols; instead, they can depend on the storage service to recover correctly after failures. Second, nearly all cloud environments offer storage services in several forms (e.g., file service, block storage) [8, 9, 10], enabling developers to build systems this way on any cloud environment.

Given this popularity, recent work attempts to improve the performance [5, 11], resource efficiency [12], and recovery time [7] of cloud-native applications. However, a more fundamental question remains unanswered: *Can cloud-native applications recover their state correctly after failures?* This thesis takes the first step toward answering this question.

To do so, we carefully study how applications fail and recover when interacting with disaggregated storage services. These services use an array of techniques to keep data durable and consistent; thus, we do *not* focus on testing the storage services themselves. Instead, we focus on common ways *applications fail when interacting with storage.* We specifically focus on scenarios where applications crash and experience network glitches. Our goal is to examine if applications can recover correctly after such commonly anticipated failures (from the state available on storage).

Our main finding is that the storage client libraries used by applications (to access storage services) exhibit a variety of *post-failure behaviors*: behaviors that lead to unexpected post-failure states on the storage service. As a simple example, writes issued by the application can be reordered if the application crashes at an inopportune moment. Such behaviors threaten recoverability, ultimately leading to disastrous outcomes such as application data loss and unavailability.

More importantly, we find that the post-failure behaviors depend upon a set of *intricate predicates* (e.g., are there other pending writes? do the writes modify overlapping byte ranges?). To exacerbate the problem, the said predicates differ wildly across storage services, resulting in different states after the same application failure. For example, an application crash may result in reordered writes on one service but not on another (e.g., GlusterFS vs. JuiceFS). The predicates also vary under different configurations of the same service.

In principle, one can test an application against a storage service (and possibly fix the problems). However, the cloud-native deployment model renders such a point approach ineffective. Specifically, container orchestration frameworks like Kubernetes support tens (if not hundreds) of backing stores and provide a standard interface (CSI [13]) to access these services. Application developers typically cannot control upon which backing store their applications will be deployed. As a result, the application may break when run against a different (untested) storage service or configuration.

Our goal in this thesis is to draw attention to a pressing problem: application recoverability in the cloud-native paradigm. To this end, in our preliminary effort, we focus on disaggregated file services and carefully study whether applications atop them can correctly recover from failures. In particular, we study four applications (LevelDB [14], RocksDB [15], Zookeeper [16], and Mercurial [17]) atop two disaggregated file systems (GlusterFS [18] and JuiceFS [19]). First, we bring to light different post-failure behaviors and the predicates upon which they arise. Then, we show that the post-failure behaviors do have real-world consequences by revealing new vulnerabilities in these applications. We further develop a tool capable of simulating post-failure behaviors on application workloads and reproducing the vulnerabilities we discovered. Finally, we outline our next steps and vision to enable cloud-native applications to correctly manage persistent state.

The organization of this thesis is outlined as follows. Chapter 2 provides an overview of a standard cloud-native deployment, along with an examination of potential failure scenarios that may arise between an application and cloud storage service. In Chapter 3, we present our approach to evaluating application recoverability within a cloud-native context, incorporating an analysis of both the storage system and the application itself. Chapter 4 delves into the predicates and post-failure behaviors displayed by the file systems under investigation. Subsequently, Chapter 5 explores the application vulnerabilities we have identified. Chapter 6 details the design and implementation of a tool we have developed to reproduce existing vulnerabilities, as well as to discover new ones. Looking ahead, Chapter 7 discusses future research directions, Chapter 8 covers related work, and Chapter 9 concludes the thesis.

CHAPTER 2: BACKGROUND

In this chapter, we explore the fundamentals of cloud-native application architecture, focusing on the interaction between applications and disaggregated storage services. We begin by providing an overview of the components and architecture involved in building cloud-native applications, such as LevelDB, using storage services like GlusterFS. Additionally, we discuss common expectations regarding application recovery after failures. As we delve into the fault model, we classify various failure scenarios and examine the impact of crashes and network failures on the reliability of cloud-native applications when interacting with storage services.

2.1 CLOUD-NATIVE APPLICATION ARCHITECTURE AND EXPECTATION

In this section, we provide a brief overview of how applications (e.g., LevelDB) work atop a disaggregated storage service (e.g., GlusterFS) in the cloud-native paradigm. We also discuss common expectations regarding application recovery after failures.

Cloud-Native Application Architecture Figure 2.1(a) illustrates a typical cloud-native application architecture. The application is deployed on a stateless virtual machine (VM) or container, while its state is persisted on a disaggregated storage service. The storage service provides strong durability and consistency guarantees to ensure data integrity and availability.

Durability in the storage service is generally achieved through replication or erasure coding techniques. To maintain data integrity on each replica, the service employs checksums and crash-consistency mechanisms. Moreover, despite replication, the storage service offers strong consistency by exposing meaningful states to the application layer.

It is worth noting that storage services can provide different data abstractions, such as files, blocks, or blobs. Regardless of the specific abstraction used, the storage service is designed to present a reliable and transparent black-box view to the applications.

Cloud-Native Application Expectation Figure 2.1(b) demonstrates an example of a cloud-native application using LevelDB and GlusterFS. In this setup, GlusterFS serves as a disaggregated file service that replicates data across multiple servers. Each server utilizes its local file system to store the data persistently.

The interaction between the application (LevelDB) and the storage service (GlusterFS) is mostly transparent. LevelDB employs standard POSIX file I/O operations, which are

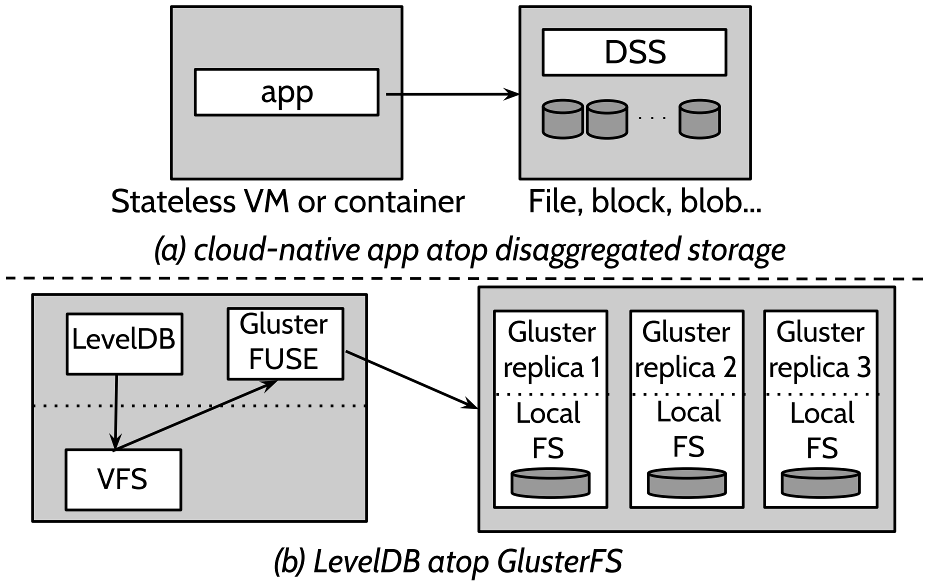


Figure 2.1: **Cloud-Native Applications.** *The figure shows how cloud-native applications are built atop disaggregated storage.*

intercepted and executed by a client-side FUSE file system [20] on the storage servers. Storage services typically have one or more client libraries that implement complex logic to interact with the storage service efficiently (e.g., buffering writes, coalescing operations, and reading data ahead of time).

A common expectation in such architectures is that the application can recover seamlessly and safely from the state persisted on the disaggregated storage service. Our objective is to examine whether this expectation holds true in current cloud-native applications and systems.

2.2 CLOUD-NATIVE APPLICATION FAULT MODEL

In this section, we discuss the fault model in the context of cloud-native applications and focus on possible failure scenarios that can arise from the interaction between applications and storage services.

Fault Model Classification Two primary categories of failures can occur in cloud-native applications when it's built a top disaggregated storage:

- Internal Storage Service Failures: These include issues within the storage service that lead to unsafe behaviors, such as data loss or corruption.

- Application-Storage Interaction Failures: These are failures that occur when the application interacts with the storage service, potentially resulting in undesirable behaviors.

Storage services employ various mechanisms, as previously discussed, to ensure data durability and consistency. Although implementation bugs can exist, extensive research has been conducted to improve the safety of storage systems (e.g., finding and fixing bugs [21, 22, 23, 24, 25, 26, 27, 28], verification [29, 30, 31]). Consequently, we assume the storage backend to be a reliable black box, which keeps the data safe and ensures strong consistency. Furthermore, we assume the storage backend to persist operations in the order submitted by the storage client. Any reordering that may occur internally on a single server replica, such as due to a local file system on that replica, is masked.

However, there has been limited research on failures that can arise at the interaction points between applications and storage services. Our study focuses on identifying and addressing these failures.

Failure Types: Crashes and Network Failures We consider two types of failures in our analysis: crashes and network failures.

- Crashes: These refer to cases where the VM or container running the application, along with the storage client (e.g., the FUSE client-side file system), crashes. The application may restart on the same or another VM/container.
- Network Failures: These are transient network failures that cause the storage client to be intermittently unable to submit operations to the storage service.

Our investigation aims to understand and mitigate the impact of these failures on the overall reliability and performance of cloud-native applications interacting with disaggregated storage services.

CHAPTER 3: METHODOLOGY

We now describe how we reason about application recoverability in the cloud-native setting. We take a two-pronged approach. First, we aim to understand the post-failure behaviors of the storage service: what post-failure states are possible on this service? What conditions lead to these states? Second, we intend to examine if applications can recover from the resulting post-failure states.

In this thesis, we focus on disaggregated file services as the storage backend. In future, we intend to explore other kinds of backends (§7). We study two widely used file systems: GlusterFS and JuiceFS. Both target cloud environments and are supported by modern platforms like Kubernetes [32] and cloud services [33, 34, 35]. They share the common architecture in Figure 2.1. We analyze the recoverability of four applications, LevelDB, RocksDB, ZooKeeper and Mercurial on the two file systems. LevelDB and RocksDB are data stores widely used in cloud VMs; ZooKeeper is a distributed coordinatior serivce; Mercurial is a version control system.

3.1 STUDYING DISAGGREGATED FILE SERVICES

Disaggregated Storage Disaggregated storage are gaining popularity in cloud computing due to their ability to provide fault tolerance and flexibility for applications. Such storage systems persist data on dedicated storage nodes that are separate from the compute node of an application. This approach enables fault tolerance by replicating data across multiple storage nodes, ensuring that the application can recover in case of storage node failures. Moreover, the separation of storage and compute nodes provides flexibility for applications. Compute nodes can access storage services from anywhere and detach or attach storage as required, allowing for efficient resource utilization.

Kubernetes is a prominent example of a platform that supports disaggregated storage patterns. By separating compute nodes, such as containers, from storage nodes like persistent volumes, Kubernetes enables the use of independent and managed disaggregated storage systems. Kubernetes provides the CSI (Container Storage Interface) [36], which allows storage providers to expose their storage systems as a backend for persistent volumes. This feature allows users to choose different storage backends for their applications by specifying different storage classes [37]. Moreover, Kubernetes enables users to configure a specific storage class to define policies that govern how storage is allocated and managed, providing greater control over the allocation and utilization of resources.

The CSI interface has led to the development of over 100 CSI drivers [38] for various disaggregated storage systems. These drivers cover a wide range of storage options, including block storage, object stores, and disaggregated file systems. This variety of options allows users to choose the most suitable storage solution for their specific needs, improving the overall efficiency and reliability of their applications.

GlusterFS and JuiceFS In this study, we undertook an analysis of disaggregated storage systems, specifically focusing on GlusterFS and JuiceFS, two widely used cloud-based file systems that can be integrated with Kubernetes through their respective CSI drivers [39, 40]. GlusterFS is a distributed file system engineered for scalability and high availability, attaining fault tolerance by replicating both data and metadata across a multitude of storage servers' local file systems. Conversely, JuiceFS leverages popular cloud object storage services, such as Amazon S3 [41], Google Cloud Storage [42], and Alibaba Cloud Object Storage [43]. Unlike GlusterFS, JuiceFS does not employ a true local file system as its backend; instead, it relies on a key-value storage layer for metadata storage and maps file system operations to the cloud object storage service.

GlusterFS and JuiceFS both adhere to a client-server architecture wherein a FUSE-like client operates on the compute nodes and a server manages storage requests. In the case of GlusterFS, the server is connected to a storage device that runs a local file system on top of it, whereas in JuiceFS, the server employs a key-value storage service for metadata transactions and the client directly communicates with the cloud storage for data persistence. Both systems convert file system operations into storage requests, which are transmitted from the storage client to the server via a reliable network channel.

Various optimization techniques are employed by GlusterFS and JuiceFS to boost performance and availability. For instance, both systems implement cache management strategies to minimize latency and enhance overall efficiency [44, 45]. The cache system comprises two components: metadata cache and data cache. The metadata cache typically stores information such as file attributes, file entries, and directories, enabling operations like `getattr` and `open` to be purely in-memory if the relevant file's metadata is already cached. The data cache is further divided into read cache and write cache; the read cache collaborates with speculative read logic to store data that the application may require in the future. The write cache is utilized to decrease write operation latency, as multiple writes can be merged and retained in a memory buffer for caching, with cached data being asynchronously uploaded to the server. Additionally, JuiceFS optimizes large files by dividing them into multiple chunks and slices, ensuring that only the relevant slice is retrieved from the cloud object storage during data manipulation [46]. Regarding availability, GlusterFS employs a synchronous replication strategy, which involves updating all copies of data on replica bricks simultaneously as a client modifies it [47]. In case a replica brick fails, the client will maintain continuous data availability by accessing data from a healthy brick. JuiceFS utilizes a consensus-based high-availability design for metadata replication. As for file data, JuiceFS's availability depends on the object storage used. For example, AWS S3 replicates data across a minimum of three Availability Zones and offers 99.99% availability [48].

Understanding Post-failure Behaviors and Predicates. Theoretically, the storage client would synchronously perform the operations on the storage service. Such a synchronous approach would largely simplify application recovery: the application can expect to recover all completed operations. Unfortunately, however, in reality, the storage client buffers writes for performance reasons.

Further, the storage client does not necessarily submit the operations *in order* to the backend. Thus, upon a crash, an operation O_2 may succeed but a prior operation O_1 may not. Similarly, a network glitch can cause an operation to fail while a subsequent operation may succeed. As a result, applications may see a variety of unexpected states after a failure. We refer to such behaviors of the storage client that lead to unexpected states as *post-failure behaviors*.

As we will show, clients of modern storage services implement a complex set of rules that determine the post-failure behavior. We refer to these rules as *predicates*. Interestingly, the predicates vary wildly across different storage clients. We describe these behaviors and the predicates in the next chapter (§4). We now describe our high-level methodology to discover these behaviors and predicates.

As we discussed, the application submits operations to a client-side FUSE file system which then submits them to the server. The client file system is the one that reorders operations. First, to examine what reorderings are possible, we run simple application workloads (e.g., insert a key-value pair) and trace the application's I/O system calls (using `strace`) and the server's I/O system calls. We then compare the traces to identify the reordering behavior. For every observed reordering, we carefully analyze the client-side file system code to understand the predicate(s) that lead to the reordering. During this inspection, we discovered more predicates that could lead to other reordering behaviors. We note two limits of our current (preliminary) approach. It is not comprehensive: it doesn't guarantee that we find all possible post-failure behaviors or predicates. Second, some steps on learning the predicates involve manual effort; we intend to enhance our tool in §6 and develop a more systematic framework to address these concerns (§7).

3.2 STUDYING CLOUD-NATIVE APPLICATIONS

Cloud-native application In our research, we define cloud-native applications as those that leverage the functionalities of the cloud environment to deliver their services. In particular, we focus on applications that use disaggregated storage services for data persistence. Our study delves into a few typical applications that fall under this category. By examining the characteristics of these applications, we aim to understand how they can benefit from the use of disaggregated storage and what challenges they are facing under the post-failure behavior of a disaggregated storage system.

LevelDB and RocksDB LevelDB [14] and RocksDB [15] are popular two widely adopted key-value stores with high performance, scalability, and ease of use. LevelDB, developed by Google, is a lightweight, embedded database that employs a log-structured merge-tree (LSM tree) data structure. RocksDB, on the other hand, is a fork of LevelDB that was developed by Facebook and optimized for solid-state drives (SSDs). RocksDB employs a similar LSM-tree data structure to LevelDB, but includes additional optimizations. Both key-value storage systems use WAL (Write-Ahead Log) to ensure durability and consistency of data in the face of system failures. Specifically, before any modifications are made to the database, a log record is written to a separate log file. This record contains the information needed to undo or redo the operation, in case of a crash or power loss. Once the log record is written, the modification can be made to the more efficient data structure which is essentially a LSM tree.

In cloud environments, LevelDB and RocksDB can benefit from the use of disaggregated storage services to enhance their fault tolerance capabilities. By replicating critical data such as WAL files across multiple storage nodes, the storage service can provide additional layers of protection against data loss or corruption due to hardware failures or other unforeseen events. In the event of a failure occurring on a single database instance, the application can be restarted on any other compute node that is connected to the disaggregated storage service. The stored data can then be recovered from the replicated WAL files on the storage service, ensuring that the system remains operational and data remains consistent. With the use of disaggregated storage, LevelDB and RocksDB can achieve higher levels of fault tolerance, flexibility, and scalability. Additionally, the separation of storage and compute resources offered by disaggregated storage allows for more efficient resource utilization, enabling applications to scale more effectively while reducing operational costs.

ZooKeeper ZooKeeper [16] is a highly available and fault-tolerant distributed coordination service that is widely used in distributed systems. Developed by Yahoo, ZooKeeper provides a simple interface for applications to coordinate and synchronize their operations in a distributed environment. Its high availability and fault-tolerance are achieved through the use of write-ahead log (WAL) and snapshot techniques. The WAL technique ensures the durability and consistency of data by recording all changes to the ZooKeeper state in a transaction log on disk before they are written to the data store. This ensures that the data can be recovered in the event of a system failure. The snapshot technique optimizes the recovery process by periodically taking a point-in-time copy of the ZooKeeper state and storing it on disk. When the system needs to be recovered, it can simply load the latest snapshot and the corresponding transaction log to reconstruct the state of the system at the time of the snapshot.

ZooKeeper offers the flexibility to operate either in standalone or distributed mode [49]. In standalone mode, the system stores its data in a local file system and functions independently without engaging with a ZooKeeper ensemble or other servers. However, in production environments, achieving high availability and fault tolerance is crucial. Distributed mode is the preferred approach, where multiple instances of ZooKeeper are deployed on different machines to form an ensemble capable of handling failures and ensuring consistency. Recent innovations in disaggregated storage have demonstrated that by utilizing a disaggregated storage service, ZooKeeper can achieve the same level of fault tolerance as distributed mode. The disaggregated storage service replicates both WAL logs and snapshots, thereby enhancing data durability and consistency in the face of hardware failures or other unexpected events. Additionally, using disaggregated storage improves resource utilization and enables better compute and storage flexibility on the ZooKeeper node.

Mercurial Mercurial[17] is a distributed version control system designed to manage source code repositories. Mercurial stores data on disk using a structure called `revlog`. The `revlog` contains a set of revisions, each representing a snapshot of the repository at a particular point in time. Each revision is stored as a delta, or a set of changes, from the previous revision. In addition to the `revlog`, Mercurial stores other metadata on disk, such as a manifest of all files in the repository, and information about branching and merging. All of this data is stored in a directory called `.hg`, which is located at the root of the repository.

In cloud-based development environments, such as GitHub.dev [50] and Gitpod [51], users are provided with a container to run and test their code, along with disaggregated storage for persisitence of code and data [52]. These environments are ideal for employing user-end version control tools like Mercurial. Upon starting a new workspace, container orchestration platforms like Kubernetes allocate a distinct container environment, equipped with essential tools such as Mercurial for version control. This isolated environment ensures that each workspace remains separate from others. Disaggregated storage enables users to maintain their code and data persistently, even when the workspace container is terminated. As a new workspace is launched, the code and data are automatically retrieved from the storage system and mounted into the workspace container. This process allows users to resume their work effortlessly. By capitalizing on disaggregated storage, Mercurial guarantees that version control data stays persistent and accessible, even during container failures or platform downtime. Furthermore, the cloud development platform enhances flexibility and minimizes costs by deallocating the container when it is not in use. Simultaneously, any changes made to the code are preserved and can be mounted to a new container in the future.

Understanding Impact on Applications. After selecting an appropriate application to run on top of a disaggregated storage service, the next step is to consider how the application recovers its state from the storage service after a restart. To achieve this, it is necessary to understand the application's persistence models by designing workloads that exercise its internal logic under different settings. For example, different persistence configurations can be tested for LevelDB, such as synchronous vs asynchronous, and altered during runtime for various inserts. Application traces are collected for each workload, and a manual analysis is conducted to identify potential vulnerabilities that may arise from post-failure behaviors, using prior knowledge of file system predicates. For instance, in RocksDB, given the knowledge that the system writes log files in a specific order, if one post-failure behavior results in write operations being persisted out of order, it may lead to recovery vulnerabilities.

However, the storage system often exhibit diverse the post-failure behaviors with complicated predicates, which requires a systematic approach to test application recovery. We now describe our high-level methodology to test application recoverability.

Given the post-failure behaviors of a storage service, our goal is to generate all possible post-failure states. We re-purpose a system-call replayer [24] to produce all such possible states on the storage service. We then restart the application to recover from each state and examine if application-level properties (e.g., acknowledged data should not be lost) hold or not. §5 describes the results of our application study, §6 describes the design and implementation of the tool.

CHAPTER 4: PREDICATES

This chapter delves into a detailed analysis of how GlusterFS and JuiceFS handle file system operations of an application, with the aim of providing insight on the factors that contribute to the file system's post-failure behaviors. Furthermore, the chapter employs the methodology discussed in §3 to examine the post-failure behaviors observed in both file systems, providing example predicates that cause such behaviors. The predicates mentioned in this chapter are not exhaustive, as these are discovered in the preliminary effort. Lastly, this chapter draws a comparison between the analyzed disaggregated file systems and local file systems.

4.1 GLUSTERFS WORKFLOW

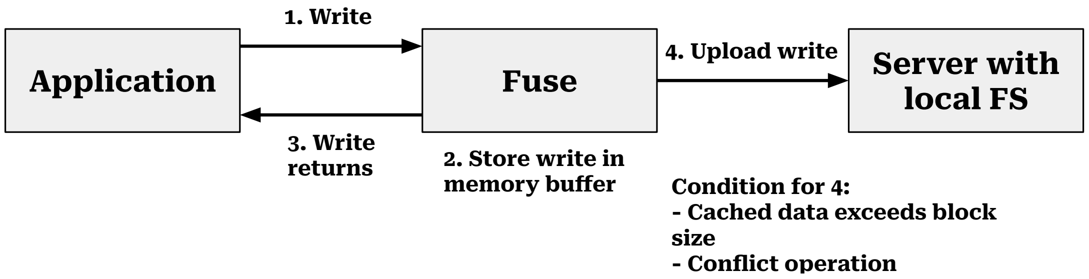


Figure 4.1: **GlusterFS Workflow.** *The figure shows how GlusterFS processes a write operation.*

In this section, we examine the GlusterFS workflow during a write operation initiated by an application, which will provide insights into the post-failure behaviors employed by GlusterFS, as discussed in subsequent sections. Figure 4.1 illustrates the process when an application issues a write on top of GlusterFS.

GlusterFS presents its clients with a FUSE-based file system that can be mounted on the application's local file system. When an application issues a write operation through a POSIX interface, GlusterFS buffers the write in memory and provides an illusion to the application that the write has completed without any errors [53]. GlusterFS employs a policy to determine when the buffered write is submitted to the server. In the default configuration, write operations are not aggregated and are sent to the server on an individual basis. When the `trickling-writes` [54] option is disabled, write operations are aggregated until they reach a configured size before being submitted to the server as a single request. In both configurations, when conflicting operations (i.e., non-write operations on the same file or write operations overlapping with the currently buffered write) occur, the buffered write is synchronously submitted to the server before the conflicting operation is executed.

Moreover, if the write transmission fails due to transient client-side faults such as network issues, the FUSE client will continue to retry the write operation in the background. If any conflicting operations arise during this process, the retry will be halted, the conflict operation will be set as failed in order to propagate the error.

4.2 JUICEFS WORKFLOW

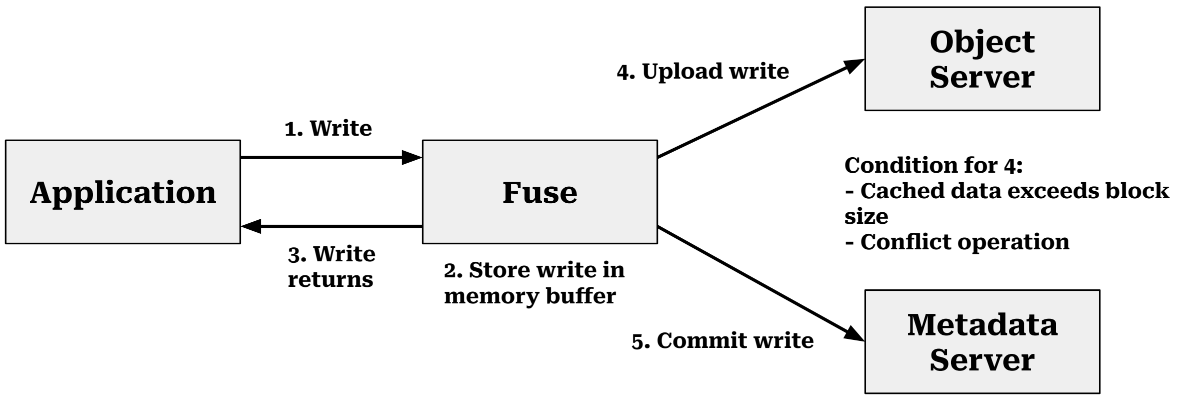


Figure 4.2: **JuiceFS Workflow.** *The figure shows how JuiceFS processes a write operation.*

In this section, we explore the underlying logic of JuiceFS, which elucidates the rationale behind the post-failure behaviors in the system. Figure 4.2 illustrates the end-to-end process when an application initiates a write operation.

By utilizing FUSE, JuiceFS facilitates POSIX-compatible access for applications, allowing them to perform file operations directly on the JuiceFS FUSE mount point. Upon receiving a write operation, JuiceFS initially stores the data in its client memory buffer and immediately acknowledges the application that the write has succeeded. The buffered data is asynchronously uploaded to the cloud object storage service for persistence when its size surpasses a predefined threshold, typically around 4MB [46]. This approach enhances overall write throughput and minimizes the number of packets transmitted between the client and server. Flush operations such as `fsync` and `close` function as ”conflict operations” in JuiceFS, compelling the buffered data to be uploaded. Notably, JuiceFS handles each operation on a per-file basis, ensuring that flush operations on one file do not impact the buffered data of other files.

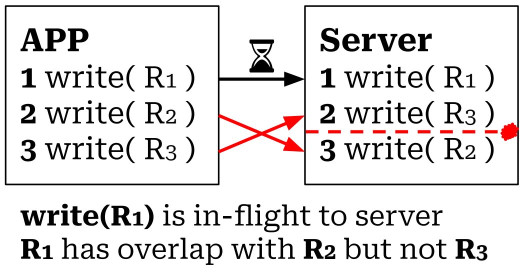


Figure 4.3: **B-1: Reordered writes to the *same file*.** *In the figure, the notation write(R) refers to a write operation performed on a specific range R within a single file.*

The actual write remains invisible to other clients until it is committed to the metadata server, which maintains a mapping between a file and all associated chunks persisted on the object server. Owing to the separation between file storage and metadata storage, file operations that exclusively manipulate the metadata server are not considered conflict operations with write operations. For instance, rename operations fall within this scope. Consequently, JuiceFS defines conflict operations differently from GlusterFS, where only operations affecting data persistence are considered.

JuiceFS also offers an "unsafe" configuration called `writeback` [55] to enhance write performance. In this mode, instead of uploading the write to the object server and subsequently committing, a commit to the metadata server is issued immediately after the buffered data is stored in an additional caching layer on the client's local file system. As a result, flush operations such as `close` and `fsync` return as successful as soon as the in-memory data is transferred to the local caching directory, eliminating the need for persistence to the server. This configuration no longer provides a safety guarantee to applications on flush operations, and an in-depth analysis of the post-failure behavior resulting from this configuration is provided in the subsequent section.

4.3 POST-FAILURE BEHAVIORS

We now describe post-failure behaviors. For each behavior, we provide example predicates that cause the behavior. We note that the behaviors and predicates are not complete: these are the ones we discovered in our preliminary effort.

B-1: Reordered Writes to the *Same File*. Writes to the *same file* issued by the application could be reordered (by the storage client). Hence, writes may not be persisted on the storage service in the same order issued by the application.

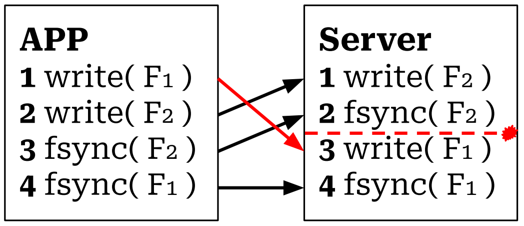


Figure 4.4: **B-2: Out-of-order writes to *different files.*** *In the figure, the notation write(F) represents a write operation performed on a specific file F.*

We observe this behavior in GlusterFS (default configuration). This behavior is dependent upon an intricate predicate used by the GlusterFS FUSE client. Specifically, when a write is in-flight, the client can issue writes that do not *conflict* (i.e., they do not touch overlapping byte ranges) with the in-flight write regardless of their order in the write buffer. Figure 4.3 illustrates this behavior. Three writes $\langle w1, w2, w3 \rangle$ on three ranges $\langle R1, R2, R3 \rangle$ are to the *same* file in the client's write buffer. $w2$ conflicts with $w1$ while $w3$ does not. Assume $w1$ is in-flight. The client could issue $w3$ before $w2$ and would issue $w2$ only after $w1$ finishes. The predicate thus ensures only a partial order; it guarantees order among conflicting writes (e.g., $w1 \to w2$) but does not guarantee order among non-conflicting writes.

This post-crash behavior will result in an unexpected state if the client crashes at an inopportune moment. In Figure 4.3, if the client crashes after it issues $w3$ but before $w2$ (as per the reordering), then it will result in a state where the later operation ($w3$) is durable while the prior one ($w2$) is not.

B-2: Out-of-order Writes to *Different Files.* Writes to *different* files issued by the application could be submitted and persisted at the remote file service out of order. Flush operations such as `fsync` and `fdatasync` only persist changes to the file specified by the input file descriptor.

We observe this behavior in both JuiceFS and GlusterFS. In JuiceFS, the client-side file system buffers the writes with *per-file* write buffers. The client does not submit a write to the remote file server, until either the corresponding write buffer is full or a conflicting operation is received (e.g., a flush operation like `fsync`). When receiving a new operation, the client-side file system checks whether it conflicts with any buffered write; if so, the client submits the previously buffered writes to the *same* file. Buffered writes to other files are not flushed to the backend. Hence, writes to different files are persisted out of order despite flush operations, as shown in Figure 4.4. This predicate is also observed in GlusterFS when `trickling-writes` is turned off, despite GlusterFS defining conflicts differently from JuiceFS.

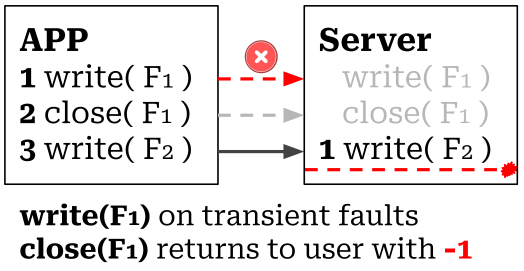


Figure 4.5: **B-3: Dropped writes on transient faults.** *In the figure, the notation write(F) represents a write operation performed on a specific file F.*

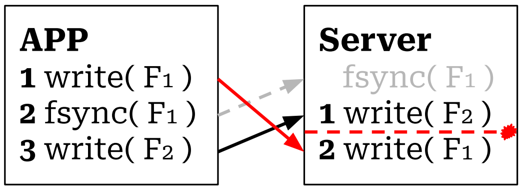


Figure 4.6: **B-4: Unsafe writes with client write cache.** *In the figure, the notation write(F) represents a write operation performed on a specific file F.*

This behavior results in an unexpected state when a failure occurs (Figure 4.4), where writes to file $F2$ is persisted and not $F1$. This behavior is error-prone when files have dependencies and writes to different files need to be ordered accordingly (see the RocksDB and ZooKeeper bugs in §5).

B-3: Dropped Writes on Transient Faults. When the client-side file system submits a write to the remote file server, the write could fail due to transient errors (e.g., socket errors due to network issues). When a write fails, all the subsequent writes that conflicts with the failed writes result in failures. These failures are not immediately exposed to the applications until a flush operation. We observe the behavior in both GlusterFS and JuiceFS under default configuration.

When the failures occur, the writes were already returned by the client-side file system to the applications with success. The failures are exposed to the application after the application issues a conflicting operation (e.g., `fsync` and `close`); see Figure 4.5. The application is expected to detect the dropped writes and handle the failures correctly. If the application overlooks the errors on subsequent conflicting operations, or is unaware of the dropped writes, it is vulnerable to data loss (exemplified by the LevelDB bug in §5).

B-4: Unsafe writes with client write cache. Writes issued by applications may not be persisted at the storage service despite flush operations. JuiceFS provides a configuration of enabling write cache at the client-side file system to improve performance of writing large amount of small files. With this configuration, writes only update the metadata service and return immediately; the data is asynchronously uploaded to the object service in the background. Flush operations such as `fsync` and `close` return with success, as soon as the in-memory data is stashed to the client write cache without submitting to the server. If the client crashes before the data in the write cache is submitted, file data is lost forever, even though the `fsync` returned success; see Figure 4.6. Moreover, writes could finish after `fsync` which can no longer guarantee the order in which writes persist. Due to the weak durability guarantees of this configuration, we will not explore its related post-failure behaviors in subsequent chapters.

Relationship to Local File Systems. Prior work [22, 24, 29] has studied the crash-consistency properties of local file systems. However, our study is different from that body of work for two reasons. First, in local file systems, the failure model is that the file system itself crashes (along with the application). Our fault model is different: we trust the storage backend to be reliable, while the application fails and recovers. Second, our study finds that a few critical properties are different from local file systems. For example, most practical local file systems persist updates to all files upon an `fsync` (i.e., the `fsync` acts as a total ordering point); in contrast, disaggregated file services persist only the `fsync`-ed file. Similarly, `close` in a local file system is an in-memory operation (with no I/O), but it acts as a `flush` in our setting [56]. Given that our preliminary study has revealed important differences in properties, we expect to uncover more differences. §6 presents our design and implementation on a preliminary tool for the new failure model, §7 discusses our next steps toward a more comprehensive study.

CHAPTER 5: APPLICATION VULNERABILITIES

We have identified new vulnerabilities in all four applications that may negatively impact their recoverability. Table 5.1 presents the post-failure behavior responsible for each vulnerability in every application, as well as the resulting consequences. The post-failure behaviors of the storage services lead to significant impacts on the availability and data integrity of all four applications. Since most of the applications we studied are databases, which play a critical role in datacenter infrastructure, these vulnerabilities can propagate and have an even greater impact on user-facing services that rely on these infrastructures. We will now discuss each vulnerability in more detail, focusing on one vulnerability in each system.

Application	Post-failure behavior	Consequence
LevelDB	B-2 and B-3	Data loss
RocksDB	B-2	Hole recovery
ZooKeeper	B-2	Hole recovery
Mercurial	B-2	Data corruption

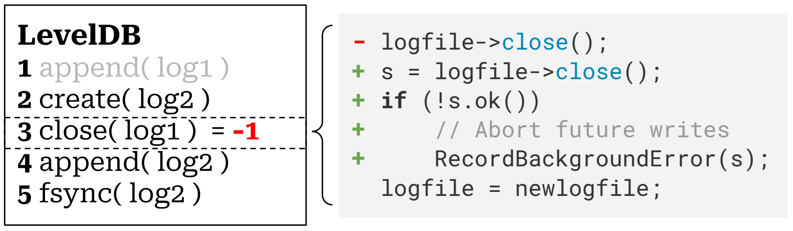
Table 5.1: Application Vulnerabilities

LevelDB Data Loss. LevelDB writes data to a log. When a log file reaches its size limit, LevelDB creates a new file and inserts data to the new one; meanwhile, it closes the old file. Figure 5.1 shows the file system operations that LevelDB issues, when it switches from `log1` to `log2`.

LevelDB was vulnerable when deployed on GlusterFS or JuiceFS. According to B-3, upon a socket error, the `append` at line 1 (L1) will fail when the client submits it to the remote file service. LevelDB can only notice the error when issuing a `close` at L3 (which returns `-1`). In GlusterFS and JuiceFS, `close` `flush`-es the file to the server. However, LevelDB does not check the return value of the `close`, losing the opportunity to handle the write failures.

Note that the `fsync` at L5 only applies to operations on `log2`, as explained in B-2. Different from many local file systems, it does not attempt to flush any writes to `log1` and thus would not expose errors with regard to `log1`.

The consequence is data loss. If a crash happens after the `fsync` at L5, LevelDB can only recover data for `log2`, but loses the data appended to `log1`. The data loss breaks the guaranteed recoverability of LevelDB with hybrid insertion mode [57] which is expected to recover the bulk writes after the last synchronous write finishes. This vulnerability is confirmed [58] and fixed by adding error checks for `close` and stopping future writes if errors occur (see Figure 5.1).



```
- logfile->close();
+ s = logfile->close();
+ if (!s.ok())
+     // Abort future writes
+     RecordBackgroundError(s);
  logfile = newlogfile;
```

Figure 5.1: **A data-loss vulnerability we found in LevelDB atop GlusterFS and JuiceFS, due to B-2 and B-3.**

Unfortunately, the fix cannot completely prevent this vulnerability when LevelDB is deployed on JuiceFS with the configuration of client-side write cache [55], in which writes are asynchronously uploaded apart from metadata.

RocksDB Holes in Recovered Data. In RocksDB, writes are recorded in both a memtable and a write-ahead log (WAL) file. When the WAL file reaches its size limit, a new WAL file is created. When the memtable is full, it is converted to an sstfile, and the corresponding WAL file is removed. Figure 5.2 shows an example of inserting four key-value pairs into RocksDB during the switch of the WAL files.

The above operations make RocksDB vulnerable when deployed on GlusterFS. As per B-2, the `append` for the second insertion (L4) is buffered and submitted to the remote storage service only when the `close` at L10 (which is a conflicting operation) occurs after log compaction. Meanwhile, the `append` for the third insertion (L6) is not reordered due to the presence of an immediate `fallocate` (L7), a conflict of the `append`. If a crash occurs *after* the first and third insertions are persisted but *before* the second insertion, and `log1` has not been compacted yet; upon restart, RocksDB will recover the first and third insertions, but lose the second insertion.

The vulnerability leads to holes in the recovered data [59]. Such holes break assumptions of many applications that use the most recent recovered write to establish the starting point for replication. In fact, RocksDB implemented a testing tool [59] to discover scenarios of holes in recovered data. However, the tool only focuses on scenarios where all writes since the last flush are lost and does not consider the potential reordering of individual writes made by the file system.

ZooKeeper Holes in Recovered Data. In ZooKeeper, updates to the key-value storage system are persistently recorded in a transaction log file. Each update is identified by a unique transaction id (`zxid`), consisting of an epoch number and a counter. The epoch number denotes a change in leadership and is increased whenever a new leader takes charge.

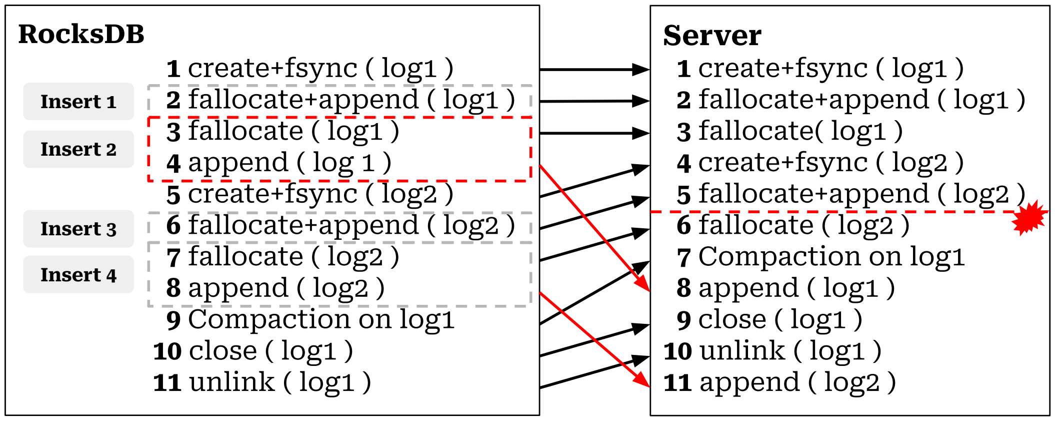


Figure 5.2: **A vulnerable sequence of operations that leads to holes in recovered data in RocksDB on GlusterFS.**

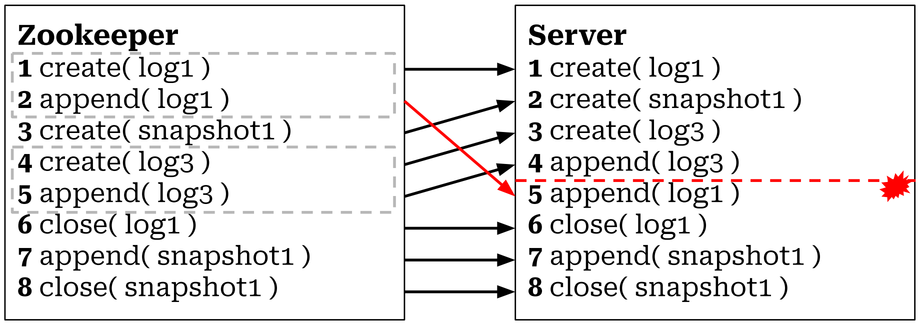


Figure 5.3: **A vulnerable sequence of operations that leads to ZooKeeper hole recovery on GlusterFS.**

Updates are appended to a log file associated with a specific epoch number, which rotates to the next epoch number when a new leader is elected or when the number of updates in the current log file reaches a preconfigured limit. Concurrently, a snapshot file of the ZooKeeper state is asynchronously created and named using the previous epoch number to indicate it includes the state up to that epoch. This process ensures the durability and consistency of the data in ZooKeeper's key-value storage system.

By default, ZooKeeper's `ForceSync` option is enabled, requiring all updates to be synchronized with the transaction log's media before completing the update processing. This involves issuing an `fsync` after every append operation to the log file, ensuring data durability. Despite this, disabling `ForceSync` can greatly enhance performance, prompting many practitioners to do so in real-world deployments where they can tolerate some recent write losses. In case of data loss, these applications rely on writes recovered by ZooKeeper to determine the starting point for resuming operations. The figure below illustrates the insertion

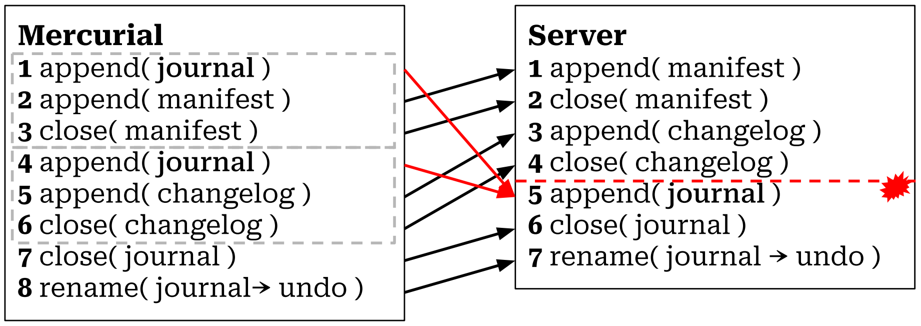


Figure 5.4: **A vulnerable sequence of operations that leads to Mercurial repository corruption on GlusterFS and JuiceFS.**

of two key-value pairs into ZooKeeper while switching transaction log files when `ForceSync` is disabled.

When ZooKeeper is deployed on GlusterFS, it is vulnerable to a hole recovery issue. According to B-2, the first insertion's (L2) append operation is buffered and transmitted to the remote storage service only when a conflicting operation (`close` at L6) takes place during the snapshot-taking process. If a crash happens after the second insertion is persisted but not the first insertion, and `log1` (which contains the first insertion) hasn't been dumped to the snapshot file yet, then upon restarting, ZooKeeper will recover the second insertion but lose the first one.

The vulnerability leads to holes in the recovered data, where recovered writes are more recent than lost writes. One possible solution to this vulnerability is to perform an `fsync` operation on the transaction log file before switching to the next log file. This ensures that all writes in the previous log file are persisted before log rotation, preventing any reordering from occurring.

Mercurial Repository Corruption. A Mercurial repository maintains critical commit-related metadata in the *manifest* and *changelog* files. During the commit process (Figure 5.4), before the actual updates are made to these metadata files, Mercurial records the modifications in a *journal* file for recovery purposes. Once all metadata updates have been made, the *journal* file is closed and renamed to an *undo* file.

The repository is vulnerable to corruption, when it is deployed on JuiceFS or GlusterFS, due to B-2. Specifically, the two `append` (L1 and L4) to the journal file are not submitted to the server until the `close` at L7, a conflicting operation. The `close` at L3 and L6 only flush writes specific to the manifest and changelog files, not the journal file. If a crash occurs after the manifest or changelog file updates are persisted but not the *journal* file, the repository will be corrupted due to the inconsistencies between the metadata files. The corruption cannot be resolved by `hg recover` [60] which assumes that all metadata file updates are recorded in the journal file. A potential fix to this vulnerability is to add an `fsync` after every `append` to the journal file to guarantee the persistence order.

CHAPTER 6: POST-FAILURE ANALYZER

6.1 PFALZ DESIGN

Pfalz (**P**ost-**f**ailure **A**na**lyz**er) is an automatic framework that can produce all possible traces that the storage client might send to the server, given the post-failure behaviors. For each such trace, Pfalz works with Alice [24] to construct all possible server states, each representing a different point of application failure. After that, an application specific checker will be run on top of the server state to check if application is able to recover correctly from that state.

In order to generate storage client traces, Pfalz leverages pre-defined file system predicates, which are implemented as Python classes and accessible through a unified interface. These predicates are designed to determine the appropriate post-failure behaviors to be made in response to various operations within an application trace.

Note that existing tools (that check local file system crash consistency [24, 61]) cannot be readily applied, because they consider a fundamentally different failure model (see §2).

Pfalz test application vulnerability with the following workflow,

- Application trace collection (§6.1.1): Pfalz initiates the process by running the application workloads and collecting the associated traces.
- Storage client trace generation (§6.1.2): After obtaining the application traces, Pfalz applies various pre-defined predicates to them, resulting in the generation of all possible storage client traces that may be issued to the server.
- Server disk state construction (§6.1.3): To generate all possible disk snapshots, Pfalz performs a prefix reconstruction on the storage client traces.
- Recovery vulnerability detection (§6.1.4): Lastly, Pfalz runs a checker on each generated disk snapshot to evaluate the application recovery behavior and detect any vulnerability.

6.1.1 Application trace collection

Pfalz begins by running a specific application workload and collecting its trace using the Alice [24] tool. During the trace collection, Strace [62] is employed to attach to the workload process and obtain per-thread level file system traces, noting that threads may

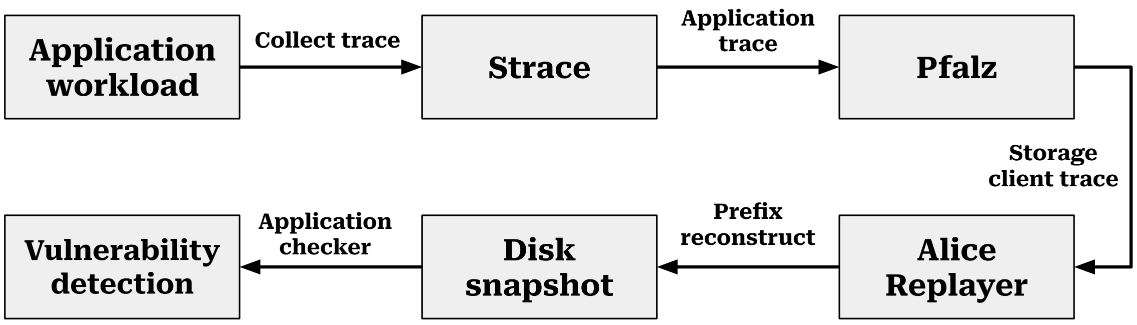


Figure 6.1: **Pfalz Pipeline.** *The figure shows Pfalz pipeline on detecting application vulnerabilities.*

share memory and file descriptor tables. To obtain an organized application level trace, Alice tracks correlations between threads' file descriptors and mmap calls, tracing each thread's `clone` and `fork` calls for this purpose. This approach enables an understanding of the file descriptor and mmap correlations between different threads. Additionally, Alice tracks each process's file descriptor and maps each `fd` to the corresponding file's inode or memory page if it is a `mmap`.

These steps result in the generation of a totally ordered trace for the analyzed application workload. This trace allows file descriptors to be mapped to their corresponding file system paths for each file system operation. The totally ordered workload trace is then used as input for the subsequent stage of the analysis.

6.1.2 Storage client trace generation

This section outlines the approach taken by Pfalz to generate traces that simulate the storage client's requests to the server. To achieve this, Pfalz relies on a prior understanding of the file system predicates to determine how to produce the storage client trace based on the application trace. The predicates are initially extracted from the file system through an in-depth analysis of the system's behavior and its code, as described in detail in §3. Subsequently, we attempt to translate the predicates into code that can be used by the Pfalz tool. As an illustration, an example for translation is presented below.

In figure 6.2, we show how predicate 2 is translated into Python code. Predicate 2 is defined as a class with a unified interface called `check_boundary`, which serves as the required entry point for every predicate. Inside this interface, the predicate code specifies what action should be taken when considering two file system operations. For example, the predicate can either be dropped or reordered from its original place to the place before the other

```python
class Predicate2:
    def check_boundary(self, cur : Op, other: Op) -> Action:
        # In B-2, we only consider write can be reordered
        if not cur.is_write():
            return Action.STOP
        # In B-2, we consider write can be rerodered after
        # any other operation that is on a different file
        if cur.file != other.file:
            return Action.CONTINUE
        # In B-2, we consider any operation on the same file
        # can be conflict, write should not be reordered after
        # conflict operation
        return Action.REORDER
```

Figure 6.2: **A translation of predicate 2.** *The figure shows predicate 2 as a Python class in Pfalz.*

operation, or it can remain unchanged in its original location. In the code snippet, we specify that the predicate will only consider actions on a write operation, and that the write operation will be reordered right before any other operation that is issued on the same file as the write.

By encoding such predicate logic inside a unified interface, Pfalz is not specific to a particular predicate and is generalizable to all file systems.

After accepting an application trace, Pfalz loops through predicate set and for each available predicate, it iterates a start operation that will be considered for manipulation by the predicate (e.g., dropped or reordered). It then iterates an end operation to check the boundary for where the start operation can be manipulated until, and this condition is checked through the unified interface exposed by that specific predicate.

For every possible manipulation, a new trace is created based on that, and all the resulting traces are collected and used as input for the next stage of analysis.

6.1.3 Server disk state construction

Upon obtaining all possible traces issued by the storage client, we consider these traces to be sent to the storage server through a reliable network channel in order. However, it is still possible for the storage client to crash during the trace issuance, resulting in various intermediate states on the server's local file system. As we assume that the client sends the trace in order, the server states are a result of a prefix of the storage client trace. Therefore, it is reasonable to perform prefix reconstruction on every possible storage client trace, where each prefix state is considered a disk snapshot that the application will later recover from.

To conduct prefix reconstruction, we employ the Alice tool [24] once again. Alice firstly creates a directory with no data, simulating the initial disk state when no file system operation has been applied. For each possible trace, Alice iterates through each file system operation in order and replays each operation on the specific directory. As each operation is replayed, it produces a possible server's disk snapshot that simulates a crash behavior after the client sends the trace to the server. All of the replayed disk snapshots are collected for vulnerability detection in the subsequent stage.

6.1.4 Recovery vulnerability detection

Following the prefix trace reconstruction, we proceed to run a checker program on each server disk state obtained. This checker program is specific to the application and is responsible for verifying a few of the application's safety properties during recovery. The checks may include whether the application can recover and restart from the state without encountering any exceptions. Additionally, further checks may be performed on the correctness of the recovered data.

Figure 6.3 shows an example of RocksDB checker program. Line 10 of the program opens the database on the disk snapshot and verifies that the return code is error-free, thus ensuring that the RocksDB can recover from the disk state without exception. The correctness of the recovered data is then checked on Line 17 and Line 18, by comparing the key-value pairs with the ones persisted by the application in the workload. This step is accomplished by iterating through all the recovered key-value pairs and verifying their correspondence with the original data. Line 26 further checks the hole recovery behavior on the resulting data, by ensuring that the recovered data is continuous and in the same order as written to the disk.

Pfalz collects checker results of each disk state and identifies any prefix state that results in checker failure. A checker failure indicates a possible application recovery vulnerability resulting from a specific file system's post-failure behavior. In the RocksDB example, the storage client trace resulting from the post-failure behavior B2 will cause hole recovery, leading to the failure of the assertion on Line 26. As a consequence, the exception will be collected, and the vulnerability will be revealed.

```cpp
int main(int argc, char *argv[]) {
        /* Variable declarations and some setup */
        DB* db;
        Options options;

        int retreived_rows = 0;
        int row_present[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

        /* Check if RocksDB can recover without exception */
        auto ret = DB::Open(options, "testdb", &db);
        assert(ret.ok());

        /* Check recovered data correctness */
        auto it = db->NewIterator(read_options);
        for (it->SeekToFirst(); it->Valid(); it->Next()) {
            int row_number = it->key().ToString().c_str()[0] - 'a';
            assert(expected_key(row_number) == it->key().ToString());
            assert(expected_value(row_number) == it->value().ToString());

            row_present[row_number] = 1;
            retreived_rows++;
         }

        /* Check hole recovery */
        for(int i = 0; i < retreived_rows; i++)
            assert(row_present[i] == 1);

        return 0;
}
```

Figure 6.3: **RocksDB checker program.** *The figure shows an example of a RocksDB checker program implemented in C++.*

6.2 PFALZ IMPLEMENTATION

We implement Pfalz tool mainly in Python 3, which includes the storage client trace generator and the predicate logic for post-failure behaviors (§6.1.2). We also utilized certain functionalities of Alice, such as `strace` parser (§6.1.1) and disk snapshot reconstruction (§6.1.3).

To facilitate the integration of Pfalz with Alice, we made some modifications to Alice's existing functionalities. Firstly, we exported necessary trace-related data structures from

Alice to a JSON file using the jsonpickle library [63] since Alice is mainly written in Python 2, while Pfalz is implemented in Python 3. This allowed us to access such data structures from Pfalz as a separate Python 3 module.

Secondly, after generating storage client traces, we exported them back to Alice using the jsonpickle library. We also added Alice the ability to import traces from external JSON files and restored them to its internal data structure.

Thirdly, Alice originally constructed a `replayer` instance for reconstructing disk state on a single trace. To account for the multiple generated storage client traces, we modified Alice to construct `replayer` instances for each trace and perform prefix reconstruction on each `replayer` instance. To avoid redundant reconstruction, we implemented a caching mechanism across the `replayer` instances, so that previously reconstructed disk snapshots could be reused, given that different client traces may still share a common prefix.

Finally, the checker is executed on each prefix disk state across multiple `replayer` instances in a multi-threaded manner. We developed an analyzer to reason about the detected vulnerability with post-failure behavior instrumented on application workload.

6.3 PFALZ RESULTS

In our preliminary experiment, we evaluated the effectiveness of Pfalz in manipulating application traces based on post-failure behaviors that we have studied and discussed in §4. Our results indicate that Pfalz can successfully reproduce all existing vulnerabilities that we have identified and discussed in §5. Specifically, we generated storage client traces by applying various post-failure behaviors to a set of studied application workloads, and Pfalz was able to detect the vulnerabilities exposed by these traces. The results of our experiment are summarized in the table below, which includes the number of storage client traces generated for each post-failure behavior and the corresponding traces that expose vulnerabilities.

Application	# B-1	# B-2	# B-3	Vulnerability Reproduced By
LevelDB	0	7	2	B-3
RocksDB	2	8	2	B-2
ZooKeeper	0	2	4	B-2
Mercurial	0	7	21	B-2 & B-3

Table 6.1: Pfalz Results Table

CHAPTER 7: DISCUSSION

We now describe our plans for future work, our vision, and the challenges involved in realizing this vision.

A Comprehensive Study This thesis is a first step towards a complete study of application recoverability in cloud-native settings. In future work, we plan to expand our analysis to include a wider range of post-failure behaviors and predicates. We will explore other disaggregated file systems, focusing on identifying predicates that are unique as opposed to local file systems. Additionally, we will consider a more diverse set of applications, such as relational databases (e.g., SQLite [64], HSQLDB [65], PostgreSQL [66]), consensus-based key-value storage systems (e.g., Redis [67], etcd [68]), message queues (e.g., Kafka [69], RabbitMQ [70]), and stream processing engines (e.g., Hadoop [71], Spark [72]). One of the challenges we face is the manual nature of several steps within our current methodology. To address this issue, we plan to enhance our automated tools for a more streamlined approach.

Enhanced Tool In our research, the developed tool successfully reproduces existing vulnerabilities, but it is not without limitations. The first limitation concerns the post-failure behavior B-3 (§4), which can lead to vulnerabilities with the assumption that an application fails to check the return value of the `close` function. Our tool currently lacks the ability to discern such application-level logic. A potential solution involves fault injection during the `close` operation while the application is running its workload (e.g., by implementing a FUSE-based fault injector). If the application's trace differs from that of a normal run (without fault injection), it can be inferred that the application checks the return value of the `close` function. The second limitation pertains to the integration of file system predicates into our tool. At present, manual analysis of the file system code is required before translating the insights into predicates coded in Python. Our goal is to develop an automated method for inferring predicates and encoding them using well-structured rules.

Moving forward, we aim to address these limitations to enhance the tool's capabilities, thereby facilitating the discovery of novel vulnerabilities in a more automated and efficient manner.

A Unified File System Interface Our study shows that post-failure behaviors vary across file services. We envision a unified interface for file services that offer meaningful post-failure behaviors, while allowing the file services to implement optimizations underneath the common interface. This would enable applications to understand all potential post-failure behaviors through a well-defined interface and design persistence protocols accordingly.

A Unified Framework for All Storage Backends Our study so far has focused on distributed file services. However, modern cloud-native applications use other storage abstractions like blobs [41], queues [73], and table [74]. We envision our tools to be agnostic of the underlying storage engine and still be able to determine post-failure behaviors and test applications. We plan to abstract away the implementations and do all our reasoning at the interface level. Specifically, we will focus on storage client behaviors with publicly available cloud storage clients (e.g., S3 SDK [75], Azure Storage SDK [76]) and minimize attention to storage server implementation details. We aim to concentrate on standard interfaces across different storage services and model storage client behaviors in a more uniform manner.

Towards Correct and Portable Cloud-native Application Our vision is to ensure the correctness of applications built in the cloud-native paradigm. We believe three (complementary) approaches could take us closer to this vision. The first and most pragmatic way would be to find and fix application vulnerabilities. One challenge here would be that fixing vulnerabilities might impact performance.

However, the first approach alone cannot guarantee that applications will be portable across storage services, a reality that applications must cope with. To enable portability, we envision a principled approach where applications could specify what behaviors they expect from storage services. Cloud providers could then use this information to provide the application with the right service.

Finally, cloud storage must offer better interfaces that enable applications to realize correctness seamlessly, without forgoing performance. On one end, a simple synchronous interface could ease correctness but cause poor performance. On the other, the current way of offering poor guarantees for high performance impairs application correctness. A middle ground may resolve this tension. Our study of applications and cloud storage could pave the way for such an approach.

CHAPTER 8: RELATED WORK

Cloud-Native Applications The rise of cloud-native applications has led to an increased reliance on cloud services for managing persistent state and compute resources [1, 2, 3, 4, 5, 6]. This approach simplifies application development by offloading durability and consistency concerns to cloud storage services, making it easier for developers to build and deploy their applications on various cloud environments. Recent work has attempted to improve the performance [5, 11], resource efficiency [12] of cloud-native applications. However, our work is the first to address the fundamental question of whether cloud-native applications can recover their state correctly after failures when interacting with disaggregated storage services.

Storage Services Correctness Disaggregated storage services, such as file systems, block storage, and object storage, are crucial for managing persistent state in cloud-native applications [8, 9, 10]. These services use a variety of techniques to ensure data durability and consistency, and applications interact with them through storage client libraries. Previous work has focused on the correctness of storage services themselves, for example, several research has presented testing and model checking tools for detecting crash-safe bugs [24, 26, 27, 28, 77], fault-tolerance and recovery techniques [78, 79, 80] for storage systems deployed on local file systems, and building formally verified storage systems [31] to achieve crash safety on local file system. However, they do not focus on the post-crash behavior specific to the cloud-native setting (§4) and cannot effectively detect or prevent the vulnerabilities (§5) that impair applications recoverability.

Recover from Disaggregated Storage Previous study has demonstrated that it is feasible and beneficial to delegate replication tasks to underlying storage services [81]. With the rise of cloud storage, research has indicated that replicated data services can leverage the built-in fault-tolerance of these services, thus reducing the cost of implementing application-level fault-tolerance [82]. To utilize disaggregated storage services for replication, it is important to enhance the recovery process after a primary failure to improve availability [7]. However, these studies do not address the issue of unexpected post-failure behaviors exhibited by the storage service that can impact the correctness and availability of the application.

Container Orchestration and Storage Container orchestration frameworks like Kubernetes have become the de facto standard for deploying and managing cloud-native applications. Numerous research efforts have investigated the challenges and potential solutions related to the reliability of container orchestration systems. In particular, some studies have concentrated on the correctness of the control plane within cluster management systems, while others have focused on application reliability when deployed inside a managed cluster. For instance, prior research has attempted to model infrastructure control plane from the perspective of partial history [83]. A subsequent study has introduced a testing tool designed to identify bugs in cluster-management controllers, as illustrated in [84]. Additionally, chaos testing has been employed to simulate faults in the cluster and assess the application's reliability when confronted with issues such as container termination, network partition, I/O delay, and read/write errors [85, 86].

Kubernetes supports a wide range of storage services through the Container Storage Interface (CSI) [38]. While this flexibility enables developers to build applications that can run on multiple cloud environments, it also introduces challenges in ensuring correct recovery and consistency across different storage services and configurations. Despite this, previous research either neglects to address the storage aspects of container orchestration systems or merely simulates generic file system faults in a coarse-grained manner. Our work emphasizes the need for a more comprehensive approach to address application recoverability when considering different kind of cloud storage options, taking into account the complexity of modern container orchestration and storage ecosystems.

CHAPTER 9: CONCLUSION

In conclusion, this research thesis highlights a critical issue regarding the recoverability of cloud-native applications after failures. The thesis explores the behavior of disaggregated storage services and their interaction with applications in the event of failure. The findings reveal that post-failure behaviors of storage services can lead to unexpected post-failure states, resulting in catastrophic outcomes such as data loss and unavailability. The study underscores the importance of recoverability in cloud-native applications and identifies the need to address this problem. Further research and action are required to ensure the reliable recovery of applications in the event of failures. This thesis outlines the next steps and vision to address this problem, which can lead to the development of effective strategies to enhance the recoverability of cloud-native applications.

REFERENCES

[1] "Cloud native computing foundation," https://www.cncf.io.

[2] F. Li, "Cloud-native Database Systems at Alibaba: Opportunities and Challenges," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2263–2272, July 2019.

[3] Purvi Desai and Kevin Leong, "Rockset Concepts, Design, and Architecture," https://rockset.com/whitepapers/rockset-concepts-designs-and-architecture.

[4] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang et al., "The Snowflake Elastic Data Warehouse," in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, June 2016.

[5] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang et al., "PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers," in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*, June 2021.

[6] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, and V. Venkataramani, "Cloud-Native File Systems," in *Workshop on Hot Topics in Cloud Computing (HotCloud'18)*, July 2018.

[7] N. Li, A. Kalaba, M. J. Freedman, W. Lloyd, and A. Levy, "Speculative Recovery: Cheap, Highly Available Fault Tolerance with Disaggregated Storage," in *2022 USENIX Annual Technical Conference (ATC'22)*, July 2022.

[8] "Cloud Storage on AWS," https://aws.amazon.com/products/storage.

[9] Microsoft, "Introduction to Azure Storage," https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction.

[10] "Storage options," https://cloud.google.com/compute/docs/disks.

[11] Q. Zhang, X. Chen, S. Sankhe, Z. Zheng, K. Zhong, S. Angel, A. Chen, V. Liu, and B. T. Loo, "Optimizing Data-intensive Systems in Disaggregated Data Centers with Teleport," in *Proceedings of the 2022 International Conference on Management of Data (SIGMOD'22)*, June 2022.

[12] Y. Zhang, C. Ruan, C. Li, X. Yang, W. Cao, F. Li, B. Wang, J. Fang, Y. Wang, J. Huo et al., "Towards Cost-effective and Elastic Cloud Database Deployment via Memory Disaggregation," *Proceedings of the VLDB Endowment*, vol. 14, no. 10, pp. 1900–1912, June 2021.

[13] Saad Ali, “Container Storage Interface (CSI) for Kubernetes GA,” https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga.

[14] S. Ghemawhat, J. Dean, C. Mumford, D. Grogan, and V. Costan, “LevelDB,” https://github.com/google/leveldb.

[15] Facebook, “RocksDB,” http://rocksdb.org.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: wait-free coordination for internet-scale systems,” in *USENIX annual technical conference*, vol. 8, no. 9, 2010.

[17] “Mercurial,” https://www.mercurial-scm.org.

[18] “GlusterFS,” https://www.gluster.org.

[19] “JuiceFS - Open Source Distributed POSIX File System for Cloud,” https://juicefs.com.

[20] “FUSE — The Linux Kernel documentation,” https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[21] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, “Cross-checking semantic correctness: The case of finding file system bugs,” in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP’15)*, Oct. 2015.

[22] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, “Finding crash-consistency bugs with bounded black-box crash testing,” in *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI’18)*, Oct. 2018.

[23] V. Nossum and Q. Casasnovas, “Filesystem fuzzing with american fuzzy lop,” in *Linux Storage and Filesystems Conference (VAULT’16)*, Apr. 2016.

[24] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications,” in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI’14)*, Oct. 2014.

[25] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, “Using model checking to find serious file system errors,” *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 4, pp. 393–423, Nov. 2006.

[26] J. Yang, C. Sar, and D. Engler, “Explode: a lightweight, general system for finding serious storage system errors,” in *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI’06)*, Nov. 2006.

[27] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Correlated Crash Vulnerabilities,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI’16)*, Nov. 2016.

[28] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. 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)*, Feb. 2018.

[29] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang, "Specifying and checking file system crash-consistency models," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Apr. 2016.

[30] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Verifying a high-performance crash-safe file system using a tree specification," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, Oct. 2017.

[31] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, "Storage Systems Are Distributed Systems (so Verify Them That Way!)," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[32] "Kubernetes," https://kubernetes.io.

[33] "Cloud Computing Services - Amazon Web Services (AWS)," https://aws.amazon.com.

[34] "Microsoft Azure: Cloud Computing Services," https://azure.microsoft.com/en-us.

[35] "Google Cloud: Cloud Computing Services," https://cloud.google.com.

[36] "Container Storage Interface (CSI) for Kubernetes GA," https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga.

[37] "Storage Classes - Kubernetes," https://kubernetes.io/docs/concepts/storage/storage-classes.

[38] "Drivers - Kubernetes CSI Developer Documentation," https://kubernetes-csi.github.io/docs/drivers.html.

[39] "A lightweight Persistent storage solution for Kubernetes / OpenShift / Nomad using GlusterFS in background," https://github.com/kadalu/kadalu.

[40] "JuiceFS CSI Driver," https://github.com/juicedata/juicefs-csi-driver.

[41] "Amazon S3 - Cloud Object Storage - AWS," https://aws.amazon.com/s3.

[42] "Cloud Storage - Google Cloud," https://cloud.google.com/storage.

[43] "Object Storage Service (OSS) - Alibaba Cloud," https://www.alibabacloud.com/product/object-storage-service.

[44] "Performance Tuning - Gluster Docs," https://docs.gluster.org/en/main/Administrator-Guide/Performance-Tuning.

[45] "Cache - JuiceFS Document Center," https://juicefs.com/docs/community/cache_management.

[46] "Architecture - JuiceFS Document Center," https://juicefs.com/docs/community/architecture#how-juicefs-store-files.

[47] "Replication - Gluster Docs," https://docs.gluster.org/en/main/Administrator-Guide/Automatic-File-Replication.

[48] "Amazon S3 FAQs — AWS," https://aws.amazon.com/s3/faqs.

[49] "ZooKeeper Getting Started Guide," https://zookeeper.apache.org/doc/r3.1.2/zookeeperStarted.html.

[50] "The github.dev web-based editor," https://docs.github.com/en/codespaces/the-githubdev-web-based-editor.

[51] "Gitpod: Always ready to code," https://www.gitpod.io.

[52] "Install a storage vendor which supports CSI snapshot in preview env," https://github.com/gitpod-io/gitpod/issues/10201, May 2022.

[53] "Write Behind Translator - Gluster Docs," https://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Developer-guide/write-behind.

[54] "Tuning Volume Options - Gluster Docs," https://docs.gluster.org/en/main/Administrator-Guide/Tuning-Volume-Options.

[55] "JuiceFS Writeback mode," https://juicefs.com/docs/community/internals/io_processing/#writeback-mode.

[56] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1st ed. Arpaci-Dusseau Books, May 2015.

[57] Jeff Dean and Sanjay Ghemawat, "LevelDB Synchronous Writes," https://github.com/google/leveldb/blob/main/doc/index.md#synchronous-writes.

[58] LevelDB-1081, "LevelDB data loss after a crash when deployed on GlusterFS," https://github.com/google/leveldb/issues/1081, Jan. 2023.

[59] Andrew Kryczka, "Verifying crash-recovery with lost buffered writes," https://rocksdb.org/blog/2022/10/05/lost-buffered-write-recovery.html, Oct. 2022.

[60] Matt Mackall, "Mercurial Commands," https://www.mercurial-scm.org/doc/hg.1.html#commands.

[61] V. Chidambaram, "Orderless and Eventually Durable File Systems," Ph.D. dissertation, University of Wisconsin-Madison, Aug. 2015.

[62] "Strace - Linux Syscall Tracer," https://strace.io.

[63] "jsonpickle Documentation," https://jsonpickle.github.io.

[64] "SQLite Home Page," https://sqlite.org/index.html.

[65] "HSQLDB," https://hsqldb.org.

[66] "PostgreSQL: The World's Most Advanced Open Source Relational Database," https://www.postgresql.org.

[67] "Redis," https://redis.io.

[68] "etcd: A distributed, reliable key-value store for the most critical data of a distributed system," https://etcd.io.

[69] J. Kreps, N. Narkhede, J. Rao et al., "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, no. 2011. Athens, Greece, 2011, pp. 1–7.

[70] "Messaging that just works — RabbitMQ," https://www.rabbitmq.com.

[71] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.

[72] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica et al., "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[73] "Introduction to Azure Queue Storage," https://learn.microsoft.com/en-us/azure/storage/queues/storage-queues-introduction.

[74] "Introduction to Table storage - Object storage in Azure," https://learn.microsoft.com/en-us/azure/storage/tables/table-storage-overview.

[75] "AWS SDK for C++," https://github.com/aws/aws-sdk-cpp.

[76] "Microsoft Azure Storage Client Library for C++," https://github.com/Azure/azure-storage-cpp.

[77] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer et al., "Uncovering bugs in distributed storage systems during testing (not in production!)," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 249–262.

[78] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Strong and Efficient Consistency with Consistency-Aware Durability," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, Feb. 2020.

[79] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Protocol-Aware Recovery for Consensus-Based Storage," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, Feb. 2018.

[80] R. Alagappan, A. Ganesan, J. Liu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* Carlsbad, CA: USENIX Association, Oct. 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/alagappan pp. 390–408.

[81] J. Kim, K. Salem, K. Daudjee, A. Aboulnaga, and X. Pan, "Database high availability using shadow systems," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 209–221.

[82] H. Saxena and J. Pound, "A cloud-native architecture for replicated data services," in *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*, 2020, pp. 19–19.

[83] X. Sun, L. Suresh, A. Ganesan, R. Alagappan, M. Gasch, L. Tang, and T. Xu, "Reasoning about modern datacenter infrastructures using partial histories," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 213–220.

[84] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic Reliability Testing For Cluster Management Controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 143–159.

[85] Bella Wiseman, "Cloudy With a Chance Of Chaos: Verifying the Resiliency Of Cloud-Native Applications," in *KubeCon North America*, Oct. 2022.

[86] "Chaos Mesh: A Powerful Chaos Engineering Platform for Kubernetes," https://chaos-mesh.org.