Towards Reliable Cloud Systems

by

Thanh D. Do

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

Doctor of Philosophy
(Computer Sciences)

at the

University of Wisconsin-Madison

2014

Date of final oral examination: 07/28/14

Committee in charge:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Haryadi S. Gunawi, Assistant Professor, Computer Sciences
Shan Lu, Assistant Professor, Computer Sciences
Menggang Yu, Associate Professor, Biostatistics

To my parents, my wife, and my wonderful kids

Acknowledgements

My Ph.D. journey has been supported greatly by faculty, friends, and family members, without whom it would have not been exceptional. I would like to express my deep gratitude for these individuals.

First and foremost, I would like to thank my three great advisors, Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, and Haryadi S. Gunawi, for their countless guidance, advice, and lessons. Remzi, the research *maestro*, showed me how one could perform top-notch research *effortlessly*. Andrea's meticulous guidance helped me sharpen many vital skills for a great researcher. Finally, Haryadi's consistent, timely, effective, and invaluable detailed advice enabled me to first survive and eventually enjoy every step of my Ph.D. journey.

I also would like to thank Shan Lu and Menggang Yu, the other members of my thesis committee, for their great insights and feedback, which helped improve the quality of the dissertation.

During my graduate life, I was fortunate to work on some projects with smart and hardworking colleagues: Samer Al-Kiswany, Mingzhe Hao, Tyler Harter, Tanakorn Leesatapornwongsa, Yingchao Liu, Lanuye Lu, Tiratat Patana-anake, Riza Suminto, and Yupu Zhang. I also enjoyed the time interacting with other students: Leo Arulraj, Vijay Chidambaram, Chris Dragga, Jun He, Ryan Johnson,

Deng Liu, Ao Ma, Sankaralingam Panneerselvam, Thanumalayan Pillai, Deepak Ramamurthi, Sriram Subramanian, Swami Sundararaman, Zev Weiss, Suli Yang, and Yiying Zhang.

My dear friends at Madison have made my graduate life more colorful and enjoyable. Special thank to Hung Duc, Thiem Hoang, Huong Huynh, Tam Le, Trang Phung, Khai Tran, Quoc Tran, Quy Vuong, and Trang Vu, for the unforgettable time we had together.

Finally, I would like to thank my family for their unconditional love and support. My Mom and Dad have always been such a great inspiration for my career; their endless love and encouragement is the key for my success. My loving wife, Hoai, has shared with me every delightful and stressful moment. I am thankful that she has always been extremely supportive, understanding, and loving. My three-year-old daughter, Julia, has always cheered me up when I am exhausted. My dear sister, Thuy Duong, has always been proud of whatever her only brother has accomplished. I dedicate this dissertation to my loving family.

Abstract

TOWARDS RELIABLE CLOUD SYSTEMS

Thanh D. Do

Although providing tremendous access to data and computing power of thousands of commodity servers, large-scale cloud systems must address a new challenge: they must detect and recover from a growing number of failures, in both hardware and software components. The growing complexity of technology scaling, manufacturing, design logic, usage, and operating environment increases the occurrence of failures. Bits, sectors, disks, machines, racks, and many other components fail; worse, they fail differently (*e.g.*, stop, corrupt, and *limp*). Moreover, it is not rare for today's cloud systems to see a series of multiple failures, *i.e.*, recovery itself can see another failure.

Unfortunately, failure handling has proven to be problematic in today's cloud systems. The failure recovery path is often complex, under-specified, and tested less frequently than the normal path. As indicated by recent cloud outage incidents, existing large-scale cloud systems are still fragile and error-prone [119]. Thus, today's systems have more responsibilities: they should anticipate not only all individual failures but also rare combinations of failures. Moreover, they must efficiently handle a variety of failures such as fail-silent and performance failures.

This dissertation begins with a simple question: why are today's large-scale cloud systems not 100% reliable? We seek the answer to this vital question by making the following contributions. First, as recovery is currently under-specified, especially when handling multiple failures, we improve the state-of-the-art of cloud recovery testing by introducing a novel framework: FATE (Failure Testing Service) and DESTINI (Declarative Testing Specifications). With FATE, recovery is systematically tested in the face of multiple failures. With DESTINI, correct recovery is specified clearly, concisely, and precisely. We apply this framework to unearth failure recovery problems in three systems: HDFS, Zookeeper, and Cassandra.

Second, to address the challenge of handling fail-silent behaviors caused by memory corruption and software bugs, we propose that cloud systems should be hardened using *selective* and *lightweight* versioning (SLEEVE). With this approach, actions performed by important parts of cloud systems are checked by a second implementation of the subsystem that uses lightweight, approximate data structures. We apply the concept of SLEEVE to harden HDFS to form HARDFS. We show that HARDFS is robust and efficient.

Third, we highlight one overlooked cause of performance failure: *limpware* – limping hardware whose performance degrades significantly compared to its specification. To measure the system-level impact of limpware, we assemble *limpbench*, a set of tests that combine data-intensive load and limpware injections, and analyze five popular and varied cloud systems. We find that many cloud systems are not limpware tolerant: limpware can severely impact distributed operations, nodes, and an entire cluster; it can cause *limplock*, a situation where a system progresses extremely slowly and is not capable of failing over to healthy components.

From these findings, we introduce PMC, a *performance model-checking* approach that leverages abstraction and the limplock property to unearth limplock bugs in complex systems effectively. In our preliminary experience, we apply the PMC approach to unearth two new limplock bugs in the Hadoop speculative execution protocol.

Finally, we advocate that future generations of cloud systems should be built to be limpware-tolerant. We hope that this dissertation helps bring new insights into rethinking reliability in the cloud and advancing the state-of-the-art of building reliable systems.

Contents

Chapter 1

Introduction

Three characteristics dominate today's large-scale computing systems. The first is the prevalence of large storage clusters. Storage clusters at the scale of hundreds or thousands of commodity machines are increasingly being deployed. At companies like Amazon, Google, Yahoo, and others, thousands of nodes are managed as a single system [33, 91]. This first characteristic has empowered the second one: Big Data. On average, a person can generate a terabyte of digital data annually [127], a scientific project can capture hundreds of gigabytes of data per day [152, 166], and an Internet company can store multiple petabytes of web data [132, 144]. This second characteristic attracts the third: large computational jobs. Web-content analysis has become popular [65, 144], scientists now run a large number of complicated queries [104], and it is becoming typical to run tens of thousands of jobs on a set of files [18, 56].

Although providing tremendous access to data and computing power of thousands of commodity servers, large-scale cloud systems must address a new challenge: they must detect and recover from a growing number of failures, in both

hardware and software components [17, 37, 100, 105]. Bits, sectors, disks, machines, racks, and many other components fail. With millions of servers and hundreds of data centers, there are millions of opportunities for these components to fail [100]. Failing to deal with failures will directly impact the reliability, availability and performance of data and jobs. Unfortunately, failure recovery has proven to be challenging in these systems. For example, Facebook lost millions of photos due to simultaneous disk failures that should rarely happen at the same time [131] (but it happened); a large bank was fined a record total of £3 millions after losing data on thousands of its customers [135]; T-Mobile Sidekick, which uses Microsoft's cloud service, also lost its customer data [128]. Bug repositories of open-source cloud software hint at similar problems [10]. These incidents have shown that existing large-scale storage systems are still fragile and error-prone.

Thus, as failure becomes the norm, when data is lost and availability and performance are reduced, we should no longer blame the failure, but rather the inability to handle the failure. Although failure-handling principles have been highlighted before [34, 59, 90, 140, 141, 149], they were proposed in much smaller settings [37]. At a larger scale, we believe today systems have more responsibilities: they should anticipate not only all individual failures but also rare combinations of failures [100]; they must efficiently handle a variety of component failures, ranging from fail-stop failures (*e.g.*, machine crashes and disk failures) to more subtle ones such as fail-silent and performance failures resulting from memory corruption, software bugs [20, 101, 151], and degraded hardware [77, 79].

A component failure, if not handled properly, may lead to a whole-system failure. Although most systems are equipped with recovery logic to handle component failures, it is possible that during recovery of the first component failure,

more failures could occur. Therefore, in this dissertation, we focus on one of the biggest challenges with failure recovery in the cloud – failures *during* recovery. Moreover, not all component failure modes are straightforward to handle. While fail-stop failures can be easily detected using timeouts and error exceptions, *fail-silent* failures, scenarios where a component just simply misbehaves instead of crashing, are much harder to deal with. Finally, a component can suffer from *performance degradation*, a failure mode that has not been well-studied in the cloud context. As a result, in this dissertation, we also focus on techniques to handle these two vexing failure modes.

These observations leave many pressing challenges for large-scale system designers: How can we verify the correctness of cloud systems in how they deal with the wide variety of possible combinations of component failures (*i.e.*, failures *during* recovery)? How can we build distributed systems that efficiently handle fail-silent failures? What are the impact of degraded hardware on today's systems? How should future-generation systems handle this vexing failure mode effectively? Answers to these questions are crucial in building reliable cloud systems. We address these challenges with the following contributions.

- First, we build a new testing framework for cloud recovery: FATE (Failure Testing Service) and DESTINI (Declarative Testing Specifications). With FATE, recovery is systematically tested in the face of multiple failures. With DESTINI, correct recovery is specified clearly, concisely, and precisely [96]. We integrated our framework to three popular cloud systems (Cassandra, HDFS, and Zookeeper), explored over 40,000 failure scenarios, wrote 74 specifications, found 16 new bugs, and reproduced 51 old bugs.

- Second, we propose *selective* and *lightweight* versioning (SLEEVE), a new approach for building systems that efficiently handle fail-silent (non fail-stop) behaviors that could result from memory corruption or software bugs. The SLEEVE principle advocates that developers selectively protect important subsets of a target system in a lightweight and approximate manner [168]. We applied SLEEVE to harden HDFS to form HARDFS; through experiments, we showed that HARDFS detects and recovers from a wide range of fail-silent behaviors caused by random bit flips, targeted corruptions, and real software bugs.

- Third, we analyze the impact of degraded hardware on five popular and varied cloud systems (Cassandra, Hadoop, HBase, HDFS, and Zookeeper) and show that slow hardware can severely impact distributed operations, nodes, and an entire cluster. We uncover many design deficiencies in these systems that can cause *limplock*, a situation where a system progresses extremely slowly and is not capable of failing over to healthy components (*i.e.*, the system is *locked* in limping mode) [79].

- Finally, we introduce PMC (Performance Model-Checking), a new approach that enables finding limplock bugs in complex systems effectively. Unlike traditional model checking that typically checks for correctness properties, PMC verifies the performance properties of cloud systems. We applied the PMC approach to unearth limplock bugs in Hadoop.

  We discuss each of these contributions in the following sections.

1.1 A Framework for Cloud Recovery Testing

As mentioned above, a critical factor in the availability, reliability, and performance of cloud services is how they react to failure. However, practitioners continue to bemoan their inability to adequately address these recovery problems. For example, engineers at Google consider the current state of recovery testing to be behind the times [57], whereas others believe that large-scale recovery remains under-specified [37]. These deficiencies demand new approaches in cloud recovery testing.

To address this challenge, we present two advancements in the current state-of-the-art of testing. First, we introduce FATE (Failure Testing Service). Unlike existing frameworks where multiple failures are only exercised randomly [57, 167, 183], FATE is designed to *systematically* push cloud systems into many possible failure scenarios. FATE achieves this by employing *failure IDs* as a new abstraction for exploring failures. Using failure IDs, FATE has exercised over 40,000 unique failure scenarios, and uncovered a new challenge: the exponential explosion of multiple failures. To the best of our knowledge, we are the first to address this in a more systematic way than random approaches. We do so by introducing novel prioritization strategies that explore non-similar failure scenarios first. This approach allows developers to explore distinct recovery behaviors an order of magnitude faster compared to a brute-force approach.

Second, we introduce DESTINI (Declarative Testing Specifications), which addresses the second half of the challenge in recovery testing: specification of expected behavior, to support proper testing of the recovery code that is exercised by FATE. With existing approaches, specifications are cumbersome and difficult to write, and thus present a barrier to usage in practice [99, 124, 125, 146, 184]. To

address this, DESTINI employs a relational logic language that enables developers to write clear, concise, and precise recovery specifications; we have written 74 checks, each of which is typically about 5 lines of code. In addition, we present several design patterns to help developers specify recovery. For example, developers can easily capture facts and build expectations, write specifications from different views (*e.g.*, global, client, data servers) and thus catch bugs closer to the source, express different types of violations (*e.g.*, data-loss, availability), and incorporate different types of failures (*e.g.*, crashes, network partitions).

1.2 Handling Fail-silent Failures with SLEEVE

Large-scale distributed storage systems [58, 87, 120, 155] often run on clusters of thousands of unreliable commodity machines and must handle all kinds of component failures, while preserving the integrity of user data and system meta data [37, 57, 58, 69, 100, 105]. At the individual machine level, two common failure modes that these systems face are machine crashes and disk failures. To deal with these failures, there is a rich body of literature describing detection and recovery mechanisms such as journaling [90], RAID [59, 141], and redundant hardware [34]. With these advancements, fail-stop machine and disk failures are no longer considered a single point of failure in many of today's cloud systems.

Although many cloud systems are able to handle fail-stop failures, they do face new challenges. First, the systems run at a large scale [69, 100, 105]; thus, failures that used to be rare (*e.g.*, memory corruption) become more frequent [101, 151]. Second, modern software is increasingly complex, and thus software bugs are becoming more common. If not handled properly, errors resulting from memory

corruption and software bugs could become a single point of failure.

Observations from real systems show that these failures can lead to transient, non-deterministic errors, and make the system exhibit *fail-silent* behaviors (*e.g.*, send corrupt messages) rather than crashing; these fail-silent errors can lead to data loss, unavailability, and prolonged debugging effort [20, 101]. For instance, in 2008, Amazon S3, a popular cloud storage service, suffered from a severe outage when corrupt state information spread throughout the system [20]. Interestingly, since the corruption happened to internal state information, messages containing that corrupt state were still intelligible, and the corruption could not be detected by an MD5 checksum on the messages.

To effectively handle fail-silent errors, we propose that distributed systems be built with *selective* and *lightweight* versioning (SLEEVE). The goal of SLEEVE is to detect silent faults in select subsystems of a target system and to do so in a lightweight manner (with little space and performance overhead). For example, a developer can pick some important functionality (*e.g.*, file-system namespace management) and protect that functionality from fail-silent behaviors by developing a second lightweight implementation of the functionality. This approach essentially transforms a target system into an efficient two-version form that can detect (and recover from) fail-silent behaviors.

Using the SLEEVE approach, we harden the Hadoop file system (HDFS) [155]. Although HDFS already contains some mechanisms for detecting and recovering from errors (*e.g.*, replication and checksums), bugs have been found in these mechanisms, and our experiments show that HDFS is still susceptible to memory corruptions. Thus, additional hardening to prevent data loss is useful. We harden three pieces of HDFS functionality: namespace management, replica manage-

ment, and the read/write protocol, creating three robust systems, called HARDFS-N, HARDFS-R, and HARDFS-D respectively.

We evaluate the effectiveness of HARDFS by injecting random bit flips, corrupting targeted fields of important data structures, and reintroducing known bugs. Our experimental results show that while HDFS silently misbehaves in many cases, HARDFS effectively isolates the faulty behavior so that it remains within a single node. In particular, HARDFS handles 90% of the fail-silent faults that result from random memory corruption and correctly detects and recovers from 100% of 78 targeted corruptions and 5 real-world bugs that we reintroduce in our code base. Since errors do not propagate to persistent storage or other nodes, previously fail-silent errors are transformed into fail-stop errors, enabling the use of standard recovery mechanisms such as failover, single-machine reboot, or execution of fsck. Furthermore, HARDFS detection can often pinpoint corrupt data structures, enabling micro-recovery that repairs small portions of corrupted state. HARDFS is able to micro-recover in seconds instead of rebooting over many hours.

1.3 Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems

In addition to fail-silent failures, performance failures are considered another big challenge in large-scale system management. Recent work has addressed many sources of performance failures such as heterogeneous systems [95, 187], unbalanced resource allocation [94, 154, 172], software bugs [110], configuration mistakes [28], and straggling tasks [22, 72].

In this dissertation, we highlight one often-overlooked cause of performance

failures: *limpware* [1] – hardware whose performance degrades significantly compared to its specification. The growing complexity of technology scaling, manufacturing, design logic, usage, and operating environment increases the occurrence of limpware. We believe this trend will continue, and the concept of performance-perfect hardware no longer holds. We have collected reports that show how disk and network performance can drop by orders of magnitude.

From these reports, we also find that unmanaged limpware can lead to cascades of performance failures across system components. For example, at Facebook, "there was a case of a 1-Gbps NIC on a machine that suddenly started transmitting at only 1 Kbps, which then caused a chain reaction upstream in such a way that the performance of the entire workload for a 100-node cluster was crawling at a snail's pace, effectively making the system unavailable for all practical purposes" [13]. We name this condition *limplock*[2], a situation where a system progresses extremely slowly due to limpware and is not capable of failing over to healthy components (*i.e.*, the system enters and cannot exit from limping mode).

These stories led us to raise the following questions: Are today's cloud systems susceptible to limplock? What are the system-level impacts of limpware on cloud systems, and how to quantify them? Why can limpware in a machine significantly degrade other nodes or even the entire cluster? Why does this happen in current system designs?

To address these questions, we assembled *limpbench*, a set of benchmarks that combine data-intensive load and limpware injections (*e.g.*, a degraded NIC

[1] In the automotive industry, the term *limp mode* is commonly used to describe a situation where the vehicle computer receives sensor signals outside its programmed specifications. The same term is often used for software systems that exhibit performance faults [112]. We adopt the same term in the context of degraded hardware.

[2] Analogous to gridlock.

or disk). We benchmarked five popular and varied scale-out systems (Hadoop, HDFS, ZooKeeper, Cassandra, and HBase). With this analysis, we unearth distributed protocols and system designs that are susceptible to limplock and show how limplock can cascade in these systems. For example, a single slow NIC can make many map/reduce tasks enter limplock and eventually make a whole Hadoop cluster in limplock.

The limpbench results show that limpware can cripple not only the operations running on it, but also other healthy nodes, or even worse, a whole cluster. To classify such cascading failures, we introduce the concepts of *operation*, *node*, and *cluster limplock*. Operation limplock is the smallest measure of limplock where only the operations that involve limpware are experiencing slowdowns. Node limplock is a more severe condition where operations that must be served by a limplocked node will be affected although the operations do not involve limpware. Finally, cluster limplock is the most severe situation where limpware makes the performance of an entire cluster collapse.

We show how these three classes of limplock can occur in our target systems. We also pinpoint system designs that allow limplock to occur. For example, we find issues such as coarse-grained timeouts, single point of performance failure, resource exhaustion due to multi-purpose threads/queues, memoryless/revokeless retries, and backlogs in unbounded queues.

Our findings show that although today's cloud systems utilize redundant resources, they are not capable of making limpware *fail in place*. Performance failures cascade, productivity is reduced, resources are underutilized, and energy is wasted. Therefore, we advocate that limpware should be considered as a *new* and important failure mode that future cloud systems should manage.

1.4 Limplock Detection with PMC

Our approach for unearthing limplock bugs presented in the last section has two limitations. First, it is time consuming: for every protocol, we need to create a microbenchmark, set up the experiment, run the benchmark, and analyze the results manually. Moreover, we had to perform white box-analysis that involves instrumenting the systems and manually debugging for the root cause. Second, we were unsure whether any more limplock bugs could be caught. For instance, in Hadoop, we only found three bugs, leading us to raise the question: are there any limplock bugs in Hadoop, beyond the three bugs we have found? We embarked on a thorough study of 5,000 issues that have been reported by the Hadoop developers and discovered a surprising result: we found in total 38 limplock issues that appear consistently throughout the development of Hadoop.

These findings strongly motivate our work: we need a more powerful approach to handle this pervasive problem. In particular, we seek to answer an important question: *how can we find (ideally all) limplock problems in complex cloud systems*? To answer this important question, we present PMC (Performance Model-Checking), a new approach that leverages model checking to find deep limplock bugs in current cloud systems efficiently.

Model checking [111] is a verification technique that has been used to catch hard-to-find *correctness* bugs in concurrent and distributed systems [68, 82, 133, 183, 184, 185]. Model checkers take a model of the system and explore *all* reachable states of that system in order to verify correctness properties such as the absence of deadlock or data loss. As a result, model checking is suitable for finding intricate errors that are difficult to find in traditional testing.

Nevertheless, one of the biggest drawbacks with using model checking is the

problem of state space explosion. That is, the state space can grow exponentially as the model gets bigger and more complicated. Thus, resources (*e.g.*, memory) required by a model checker can quickly grow beyond the capacity of modern machines. Moreover, *accurately* modeling a system, especially its performance aspect, is challenging [36, 115].

In PMC, we address these problems by using *abstraction* and leveraging the *limplock property*. Abstraction is a well-known technique that has been used in model checking to simplify the model of a system, making model checking feasible [118]. Using abstraction, we can construct a model that captures only details (*e.g.*, protocols and data structures) that matter, thus significantly simplifying the model. Moreover, by leveraging the limplock property, we do not need accurate performance modeling, as limplock bugs can be caught using *relative* performance comparison. In our preliminary experience, we apply PMC to catch limplock bugs in an interesting case study - the Hadoop speculative execution protocol. Our results are promising: we found two *new* limplock bugs that were not found by our previous approach. Moreover, it takes only about an hour for our model checker to run and catch the bugs.

1.5 Summary of Contributions / Outline

Bellow, we summarize the contributions and present the outline for the rest of this dissertation.

- **Background**: Chapter 2 provides a background on large-scale cloud systems together with failures they need to handle.
- **A cloud recovery testing framework**: Chapter 3 presents FATE and DESTINI, a new framework for cloud recovery testing. The framework improves

the state-of-the-art of recovery testing by providing an innovative way for developers to test the recovery code of cloud systems systematically against multiple failures and to write clear and concise recovery specifications. This framework serves as the first major contribution of the dissertation.

- **A new approach to handle fail-silent failures**: Chapter 4 presents SLEEVE, an efficient approach to harden cloud systems against fail-silent failures, and its application to HDFS. Unlike traditional and heavy-weight approaches, SLEEVE selectively protects important functions of cloud systems in a light-weight manner and leverages their existing crash-tolerant mechanisms to recover. The concept of SLEEVE and its application to HDFS serve as the second major contribution of this dissertation.

- **An analysis on the impact of limpware and a new approach for limplock detection**: Chapter 5 presents the first study on the impact of limpware on today scale-out systems. The study shows that unmanaged limpware can lead to limplock, a severe performance degradation. Chapter 6 introduces PMC, a new model checking approach to unearth deep limplock bugs in distributed systems. Both the limplock study and the PMC approach serve as the third major contribution of this dissertation.

- **Related work**: Chapter 7 summarizes related research efforts.

- **Conclusion and Future Work**: Chapter 8 summarizes this dissertation, highlights lessons learned, and presents future work.

Chapter 2

Background

In this chapter, we provide the background information that motivates the need for this dissertation. Specifically, we present an overview of today's large-scale cloud systems (Section 2.1) and the challenge of handling different failure modes in the cloud (Section 2.2).

2.1 Large-Scale Cloud Systems

The field of computing is changing. In the past, improving computational performance of a single machine was the key to improving application performance. Today, with the advent of large-scale computing, improving application performance can be as simple as adding more machines. At companies like Amazon, Google, Yahoo, and others, clusters of thousands of nodes are used to run a broad range of *Big-Data* applications that collect and analyze data at a massive scale [33, 91]. These thousand-node clusters are managed by large-scale cloud systems, which play a critical role in providing users with massive data access and computation.

Google, perhaps, is the most visible example of large-scale computing. Over the last ten years, Google has published several high-quality papers that describe the design of their large-scale cloud systems. For instance, the Google storage platform is managed by GFS, a scalable distributed file system [87], which provides fault tolerance while running on inexpensive commodity hardware and delivers high aggregate performance to a large number of clients. To harness parallel computing power of their server farms, Google uses MapReduce, a computing framework that enables programmers without any experience with parallel and distributed systems to easily utilize resources of a large distributed system [72]. Users only need to specify a *map* function that processes key/value pairs to create a set of intermediate key/value pairs, and a *reduce* function that merges the intermediate values that share the same key to combine the derived data; the run-time system handles details of data partitioning, scheduling, load balancing, and error recovery. To manage structured data, Google introduces BigTable, a system that provides capabilities similar to those seen in database systems, but is specially designed to scale to petabytes of data across thousands of commodity servers [58]. BigTable uses GFS to store its log and data files, while relies on a highly available and persistent distributed lock service called Chubby for coordination [53].

Although papers from Big-Data giants such as Google and Amazon present numerous important concepts for large-scale system designs, they are fairly secretive about many specific details. Moreover, cloud systems from these companies are typically closed-source, making it challenging for interested researchers to perform further analysis. Fortunately, many open-source versions of these systems are available. In this thesis, we examine five large-scale and open-source systems, namely HDFS, Hadoop, HBase, Zookeeper, and Cassandra [2, 4, 107, 120, 155].

We choose these systems for examination because of the following reasons. First, they represent a wide range of large-scale cloud systems that are typically deployed in many places (HDFS, Hadoop, HBase, and Zookeeper are similar to Google's GFS, MapReduce, BigTable, and Chubby respectively; Cassandra is similar to Amazon's famous key-value system Dynamo [73]). Second, even though being open-source software, these systems are fairly mature and full of functionality. Finally, each system comes with a publicly-available issue repository that contains bug reports, patches, and discussion by the developers regarding the issues. These repositories are invaluable for studying challenging problems that the systems have been encountering.

We now highlight parts of this dissertation that relate to these five large-scale cloud systems. In Chapter 3, we introduce FATE and DESTINI, an innovative cloud recovery testing framework to unearth recovery problems in HDFS, Cassandra, and Zookeeper . Although our results are specific to these systems, the approach we use is general and applicable to other systems as well. In Chapter 4, we apply the SLEEVE approach to harden HDFS against fail-silent behaviors caused by memory corruption and software bugs. Again, the SLEEVE approach is general, even though our implementation is specific to HDFS. In Chapter 5, we analyze the impact of degraded hardware on all five systems. Finally, in Chapter 6, we apply the PMC approach to find limplock bugs in Hadoop.

2.2 Failures In The Cloud

Although providing tremendous access to data and computing power of thousands of commodity servers, large-scale cloud systems must address a new challenge:

Failure	Description
0.5 overheating	Power down most machines in 5 mins, 1-2 days to recover.
1 PDU failure	500-1000 machines suddenly disappear for 6 hours.
1 network rewiring	Rolling 5% of machines down over 2-day span.
1 rack move	500-1000 machines powered down, 6 hours to come back.
20 rack failures	40-80 machines instantly disappear for 1-6 hours.
5 racks go wonky	40-80 machines see 50% packet loss.
8 network maintenances	4 might cause 30-minute random connectivity losses.
12 router reloads	Takes out DNS and external vips for a couple minutes.
3 router failures	Have to immediately pull traffic for an hour.
1000 machine failures	Entire machine failures.
Thousands of disk failures	Whole disk failures.

Table 2.1: **Failures at Google Clusters.** *This table describes failures during the first year in operation of a typical cluster at Google [70].*

they must detect and recover from a growing number of failures, in both hardware and software components. The growing complexity of technology scaling, manufacturing, design logic, usage, and operating environment increases the occurrence of failures. Table 2.1 describes failures that happen to a typical Google cluster during its first year [70]. One can observe that many components such as disks, machines, routers, and racks can fail. Worse, they fail more and more often at a large scale [31, 150, 151]. Moreover, failure can be either transient (*e.g.*, temporary loss of network connectivity) or permanent (*e.g.*, whole machine failures due to power loss). As a result, it is a real challenge for large-scale cloud systems to operate correctly under these diverse and high-frequency failures.

A component failure, if not handled carefully, may lead to a whole-system failure. Although most systems are equipped with recovery logic to handle component failures, it is possible that during recovery of the first component failure, more failures could occur. Moreover, not all component failure types are straight-

forward to handle. Unfortunately, failure handling has proven to be problematic in today's cloud systems. Recent incidents have shown that existing large-scale storage systems are still fragile and error-prone [131, 128, 135]. Thus, today's large-scale cloud systems have more responsibilities: they should anticipate not only all individual failures but also rare combinations of failures (*i.e.*, failure *during* recovery) [100]; moreover, they must efficiently handle a variety of component failures. In this dissertation, we distinguish three component failure modes that cloud systems have to handle: fail-stop, fail-silent, and performance failures. Below, we explain each of these failure modes and their connections to our contributions in this dissertation.

First, large-scale cloud systems must handle *fail-stop* failures, situations where system components *stop* operating completely and permanently. Examples of fail-stop failures are machine crashes, whole-disk failures, and network outage. This type of failure is typically detected using timeouts and error exceptions. The challenge arises when the code base of cloud systems gets more complex over time, and so does their recovery code. Unfortunately, the recovery code is often complicated and under-specified [57]. Moreover, as hinted at by the data in Table 2.1, it is not rare for large-scale cloud systems to see a series of multiple failures, *i.e.*, the failure recovery itself can see another failure. As a result, we focus on one of the toughest problems of failure recovery – failure *during* recovery. It is beneficial to be able to test the recovery code of cloud systems in the face of multiple failures systematically and efficiently. We address this challenge in Chapter 3.

Moreover, in addition to fail-stop failures, a component can suffer from fail-silent failures, which are much more challenging to deal with. Faults resulting from undetected memory corruption and software bugs can lead to transient, non-

deterministic errors, which can make a component exhibits *fail-silent* behaviors (*e.g.*, send corrupt messages rather than crashing) [101]. These fail-silent behaviors are not detected, hence not handled; the systems simply misbehave, causing user frustration, degraded performance, and prolonged debugging efforts. For instance, a single bit corruption in gossip messages made Amazon S3 storage service suffer from an 8-hour outage followed by manual recovery [20]. We address the challenge of handling fail-silent failure effectively in Chapter 4.

Finally, in addition to fail-stop and fail-silent failures, today's large-scale cloud systems must handle *performance* failures, situations where a component just simply degrades in performance instead of crashing or misbehaving [77, 78, 79]. Unfortunately, degraded components are often overlooked, and if not handled properly, they can cause severe impact such as the collapse in performance of an entire cluster. For instance, at Facebook, "there was a case of a 1-Gbps NIC on a machine that suddenly started transmitting at only 1 Kbps, which then caused a chain reaction upstream in such a way that the performance of the entire workload for a 100-node cluster was crawling at a snail's pace, effectively making the system unavailable for all practical purposes" [13]. We perform a deep analysis regarding the impact of degraded hardware on today's cloud systems in Chapter 5.

In the rest of this chapter, we discuss some of the sources that lead to these failure modes and how often they occur.

2.2.1 Fail-stop Failures

Fail-stop failures can be caused by many reasons such as network outages, power loss, and disk failures. Here, we explain disk failures in detail as it is one of the major reasons that cause fail-stop failures.

The magnetic medium of the disk can have imperfections, leading to *head crashes*, where the drive head contacts the surface momentarily. A medium scratch could occur when a particle is trapped between the drive head and the media, leading to permanent failure of individual disk blocks [145]. Disk can also fail because of mechanical reasons. A drive motor can spin irregularly or fail completely. Erratic arm movements can cause head crashes and media flaws. Inaccurate arm movement caused by rotational vibration can misposition the drive head during writes, making the disk blocks inaccessible.

Many studies of disk failures show that disks can fail either completely or partially. A study of 100,000 disks over a period of five years by Schroeder et al. showed that the average percentage of disks failing per year is around 3% [150]; the authors also found that a set of disks from a manufacturer had a 13.5% failure rate. Google also released a similar rate: 8.6% [143]. Bairavasundaram et al. found that a total of 3.45% of 1.53 million disks in production systems developed latent sector errors [30]; some bad disk drives had more than 1000 errors.

2.2.2 Fail-silent Failures

Fail-silent failures could be caused by three primary sources: disk corruption, memory corruption, and software bugs.

Disk Corruption: Disk corruption happens when one reads a block of data from the disk and receives unexpected contents. This can happen for various reasons. The classic reason is the *bit rot* in magnetic media, which occurs when the magnetism of a single bit or a few bits are flipped. This type of problem can often (but not always) be detected and corrected with low-level ECC embedded in the drive. Bugs in highly complex disk controller codes can also cause block corruption.

For instance, a *lost write* bug makes the disk report that a write has completed but in fact it was never written to the disk [162]; a *misdirected write* bug makes the controller write data to the disk but to the wrong location [175]. Finally, software bugs in device drivers and file systems can corrupt disk data. For instance, buggy device drivers can issue disk requests with bad parameters or data [62, 81].

After much anecdotal evidence of disk corruption has been circulated [34, 162, 175], studies on large population of disk drives have been conducted. For instance, in a study of 1.5 million drives over three years, Bairavasundaram et al. found that nearly 1% of SATA drives exhibited corruption [31]. Specifically, they found that 400,000 blocks had checksum mismatches, creating the possibility of real data loss.

Memory Corruption: Memory errors can be induced by faulty memory chips. A faulty RAM chip might flip random bits which might be transient or permanent. Permanent or hard errors represent physical damage to the chip. Soft or transient errors might disappear upon retry or when a new value is written to that location. Memory corruption can also be caused by radiation mechanisms. Researchers have found that an alpha particle penetration can cause a random, single-bit error [130], cosmic rays can disrupt electronic circuits [189], and atmospheric neutrons can cause single event upsets (SEU) in memories [134]. Finally, memory corruption can also happen due to software bugs in systems written in unsafe languages like C and C++. Dangling pointers, buffer overflows, and heap corruption can all result in memory corruption [35].

Evidence of memory corruption has been showed by many studies. Gorman *et al.* collected data from a vast storehouse at IBM over 15-year period and confirmed the presence of errors in RAM and that the error rates correlate with eleva-

tion [136]. In a more recent study of memory errors in a large fleet of commodity servers over a period of 2.5 years [151], Schroeder et al. observed that DRAM error rates are orders of magnitude higher than previously reported: 25,000 to 70,000 errors per billion device hours per Mbit. They also observed that more than 8% of DIMMS are affected by errors per year. In a subsequent study covering 300 terabyte-years of main memory, Hwang et al. showed that a large fraction of DRAM errors in the field can be attributed to hard errors [108]. They also provided a detailed analytical study of their characteristics.

Software bugs: Finally, large-scale cloud systems are getting increasingly complex, and thus software bugs are becoming more common. Bug reports from open-source cloud systems show that a huge number of software bugs are reported daily [6, 7, 8, 10, 11]. If not handled properly, errors resulting from software bugs could cause fail-silent behaviors and become a single point of failure.

2.2.3 Performance Failures

In addition to fail-stop and fail-silent failures, hardware can suffer from performance degradation. We term this condition *limpware* - degraded hardware whose performance degrades significantly compared to its specification. To the best of our knowledge, there is no public large-scale data on limpware occurrences. Nevertheless, we have collected from practitioners many anecdotes of degraded disks and network components, along with the root causes and negative impacts [77]. These stories reaffirm the existence of limpware and the fact that hardware performance failures are not hidden at the device level but are exposed to applications.

Disks: Due to the complex mechanical nature of disk drives, disk components wear out and exhibit performance failures. For example, a disk can have a *weak head* which could reduce read/write bandwidth to the affected platter by 80%

or introduce more than 1 second latency on every I/O [75]. Mechanical spinning disks are not immune to *vibration* which can originate from bad disk drive packaging, missing screws, constant "nagging noise" of data centers, broken cooling fans, and earthquakes, potentially decreasing disk bandwidth by 10-66% [84, 103]. The disk stack also includes complex controller code that can contain *firmware bugs* that degrade performance over time [153]. Finally, as disks perform automatic *bad sector remapping*, a large number of sector errors will impose more seek cost. We also hear anecdotes from practitioners. For example, media failures can force disks to re-read each block multiple times before responding [14], and a set of disk volumes incurred a wait time as high as 103 seconds, uncorrected for 50 days, affecting the overall I/O performance [16]. We ourselves have experienced an impact of limpware; Emulab encountered an erratic RAID controller on a boss node that crippled the testbed.

Network: A *broken module/adapter* can increase I/O latency significantly. For example, a bad Fibre Channel passthrough module of a busy VM server can increase user-perceived network latency by ten times [12]. A broken adapter can lose or corrupt packets, forcing the firmware to perform *error correcting* which could slow down all connected machines. As a prime example, Intrepid Blue Gene/P administrators found a bad batch of optical transceivers that experienced a high error rate, collapsing throughput from 7 Gbps to just 2 Kbps; as the failure was not isolated, the affected cluster ceased to work [15]. A similar collapse was experienced at Facebook, but due to a different cause: the engineers found a *network driver bug* in Linux that degraded a NIC performance from 1 Gbps to 1 Kbps [13]. Finally, *power fluctuations* can also degrade switches and routers [80].

Processors and Memory: Changes in *silicon technology scaling* and scaling op-

erating voltages to *near-threshold* for energy efficiency will produce hardware with high failure rates and variable performance [41]. Processor and memory degradation can also be attributed to *aging transistors*, and might require new solutions at the architecture or system level. Poorly designed thermals, fan failures, obstructions to airflow, and challenging workload mix can lead to *overheated* processors and memory, which can cause bandwidth throttling [123]. Finally, manufacturing variation in leakage and compensating power capping can produce as much as 26% variation in deployed systems [161].

2.3 Conclusion

It is clear that failures in the cloud are complex, diverse, and frequent, and can cause a real challenge for today's large-scale cloud systems. On one hand, these systems need to constantly provide high reliability, availability, and performance to users; on the other hand, they have to effectively handle frequent and complex component failures. Unfortunately, failure-handling problems still take place, causing data loss, reduction in availability and performance, and prolonged debugging efforts. This situation strongly motivates the need for this dissertation.

Chapter 3

FATE and DESTINI: A Framework for Cloud Recovery Testing

As failure becomes the norm in cloud computing, we believe a key aspect of system design that must be scrutinized more than ever before is *failure handling*. That is, how a system reacts and recovers upon detecting failures. Specifically, today's large-scale cloud systems should anticipate not only all individual failures but also rare combinations of failures [100]. Nevertheless, failure recovery has proven to be challenging in these systems, causing severe consequence such as data loss, unavailability, and prolonged debugging effort [20, 101, 128, 131]. Moreover, practitioners continue to bemoan their inability to adequately address these recovery problems [37, 57].

In this chapter, we ask an important question: How can we verify the correctness of cloud systems in how they deal with the wide variety of possible failure modes? To answer this question, we present a novel testing framework for cloud recovery: FATE (Failure Testing Service) and DESTINI (Declarative Testing Spec-

ifications). With FATE, recovery is systematically tested in the face of multiple failures. With DESTINI, correct recovery is specified clearly, concisely, and precisely. First, we dissect recovery problems in more detail (Section 3.1). Next, we define our concrete goals (Section 3.2), and present the design and implementation of FATE (Section 3.3) and DESTINI (Section 3.4). We close this chapter with evaluation (Section 3.5).

3.1 Recovery Problems

This section presents a study of recovery problems through three different lenses. First, we recap accounts of issues that cloud practitioners have shared in the literature (Section 3.1.1). Since these stories do not reflect details, we study bug/issue reports of modern open-source cloud systems (Section 3.1.2). Finally, to get more insights, we dissect a failure recovery protocol (Section 3.1.3).

3.1.1 Lens #1: Practitioners' Experiences

As well-known practitioners and academics have stated: "the future is a world of failures everywhere" [88]; "reliability has to come from the software" [69]; "recovery must be a first-class operation" [66]. These are but a glimpse of the urgency of the importance of failure recovery as we enter the cloud era. Yet, practitioners still observe recovery problems in the field. The engineers of Google's Chubby system, for example, reported data loss on four occasions due to database recovery errors [53]. In another paper, they reported another imperfect recovery that brought down the whole system [57]. After they tested Chubby with random multiple failures, they found more problems. BigTable engineers also stated that

Problems	Count	Definitions and Examples
Incorrect	68 (75%)	Recovery exists but it is still incorrect.
Absent	14 (15%)	Unanticipated failures (*e.g.*, undetected corruption).
Coarse	7 (8%)	A bad disk (out of many) shuts down a node.
Late	2 (2%)	A failure not being detected/notified directly.
Implications	**Count**	**Definitions and Examples**
Data loss	13 (14%)	Unrecoverable data loss (*e.g.*, loss of metadata or blocks).
Unavailability	48 (53%)	Inaccessible blocks/nodes, failed jobs/operations.
Corruption	19 (21%)	Data is altered unexpectedly but still accessible.
Unreliability	8 (9%)	A corrupt replica is not timely replaced with good ones.
Performance	3 (3%)	Increased latency or reduced bandwidth.

Table 3.1: **HDFS Recovery Problems and Implications.** *This table shows the results of bug/issue study in HDFS from 2006 to 2010. In total, we found 91 issues that pertain to recovery problems related to hardware failure.*

cloud systems see all kinds of failures (*e.g.*, crashes, bad disks, network partitions, corruptions, *etc.*) [58], which other practitioners also agree with [57, 69]. They also emphasized that, as cloud services often depend on each other, a recovery problem in one service could permeate others, affecting overall availability and reliability [58]. To conclude, cloud systems face *frequent*, *multiple* and *diverse* failures [37, 57, 58, 69, 100]. Yet, recovery implementations are rarely tested with complex failures and are not rigorously specified [37, 57].

3.1.2 Lens #2: Study of Bug/Issue Reports

These anecdotes hint at the importance and complexity of failure handling, but offer few specifics on how to address the problem. Fortunately, many open-source cloud projects (*e.g.*, ZooKeeper [107], Cassandra [120], HDFS [155]) publicly share in great detail real issues encountered in the field. Therefore, we performed an in-depth study of HDFS bug/issue reports [10]. There are more than 1300

issues spanning 4 years of operation (April 2006 to July 2010). We scan all issues and study the ones that pertain to recovery problems due to hardware failures. For each relevant issue, we classify the recovery problem into four categories: incorrect (*e.g.*, recovery exists but still incorrect), absent (*e.g.*, a particular failure such as metadata corruption is unanticipated), coarse grained (*e.g.*, a faulty disk out of many in a single machine leads to whole machine shutdown), and late (*e.g.*, a failure not being detected or notified directly). We also classify implications of each recovery problem. Table 3.1 summarizes our findings.

Beyond these quantitative findings, we also made several observations. First, most of the internal protocols already anticipate failures. However, they do not cover all possible failures, and thus exhibit problems in practice. Second, the number of reported issues due to multiple failures is still small; the developers only had reported 3 issues, which mostly arose in live deployments rather than systematic testing. Finally, recovery issues appear not only in the early years of the development but also recently, suggesting the lack of adoptable tools that can exercise failures automatically.

3.1.3 Lens #3: Write Recovery Protocol

Given so many recovery issues, one might wonder what the inherent complexities are. To answer this, we dissect the anatomy of HDFS write recovery. As a background, HDFS provides two write interfaces: write and append. There is no overwrite. The write protocol looks simple, but when different failures come into the picture, recovery complexity becomes evident. Figure 3.1 shows the write recovery protocol with three different failure scenarios.

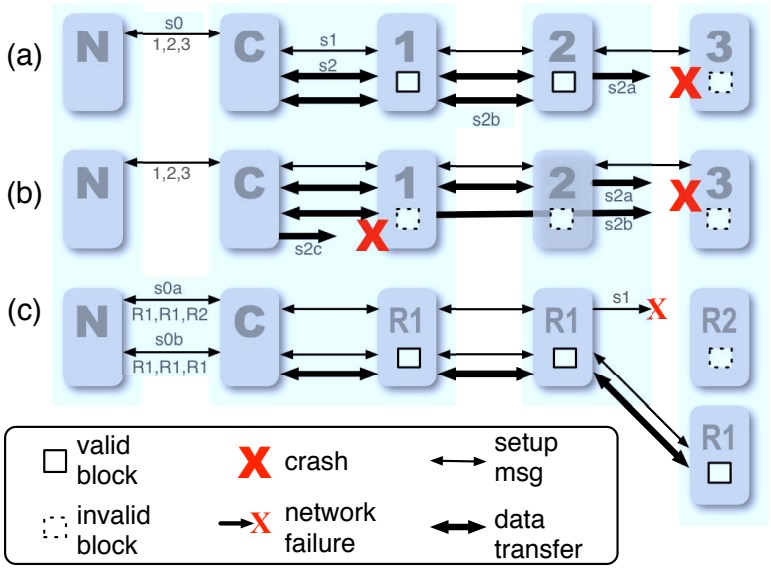

Figure 3.1: **HDFS Write Recovery Protocol.** *`N`, `C`, `R1/2`, and numeric letters represent the namenode, client, rack number, and datanodes respectively. The client always starts the activity to the namenode first before to the datanodes.*

- **Data-Transfer Recovery:** Figure 3.1a shows a client contacting the namenode to get a list of datanodes to store three replicas of a block (`s0`). The client then initiates the setup stage by creating a pipeline containing the nodes through which the setup message is sent (`s1`). After the client receives setup acks from all the nodes, it starts the data transfer stage and waits for transfer acks from all the nodes (`s2`). However, within this stage, the third node crashes (`s2a`). What Figure 3.1a shows is the correct behavior of data-transfer recovery. That is, the client recreates the pipeline by excluding the dead node and continues transferring the bytes from the last good offset (`s2b`); a background replication monitor will regenerate the third replica in the future. The design decision behind this "continue-on-surviving-nodes" approach (vs. creating a fresh 3-node pipeline) is that

the client cannot retransfer a big block (*e.g.*, tens of MB) through a fresh pipeline from the beginning because it only has a sliding window cache (5 MB by default).

- **Data-Transfer Recovery Bug:** Figure 3.1b shows a bug in the data-transfer recovery protocol; there is one specific code segment that performs a bad error handling of failed data transfer (`s2a`). This bug makes the client wrongly exclude the good node (Node2) and include the dead node (Node3) in the next pipeline creation (`s2b`). Since Node3 is dead, the client recreates the pipeline only with the first node (`s2c`). If the first node also crashes at this point (a multiple-failure scenario), no valid blocks are stored. This implementation bug reduces availability (*i.e.*, due to unmasked failures). We also found data-loss bugs in the append protocol due to multiple failures (Section 3.5.2).

- **Setup-Stage Recovery:** Finally, Figure 3.1c shows how the setup-stage recovery is different than the data-transfer recovery. Here, the client first creates a pipeline from two nodes in Rack1 and one in Rack2 (`s0a`). However, due to the rack partitioning (`s1`), the client asks the namenode again for a new fresh pipeline (`s0b`) (vs. the continue-on-surviving-nodes approach). The reason is that the client has not transferred any bytes, and thus could start streaming from the beginning. After asking the namenode in several retries (not shown), the pipeline contains only nodes in Rack1 (`s0b`). At the end, all replicas only reside in one rack, which is correct because only one rack is reachable during write [155].

- **Replication Monitor Bug:** Although the previous case is correct, it reveals a crucial design bug in the background replication monitor. This monitor unfortunately only checks the number of replicas but *not* the locations. Thus, even after the partitioning is lifted, the replicas are not migrated to multiple racks. This design bug greatly reduces the block availability if Rack1 is completely unreachable (more in Section 3.4.2).

To sum up, we have illustrated the complexity of recovery by showing how different failure scenarios lead to different recovery behaviors. There are more problems within this protocol and other protocols. Without an appropriate testing framework, it is hard to verify recovery correctness; in one discussion of a newly proposed recovery design, a developer raised a comment: "I don't see any proof of correctness. How do we know this will not lead to the same or other problems? [10]"

3.2 Goals

To address the aforementioned challenges, we present a new testing framework for cloud systems: FATE and DESTINI. We first present our concrete goals here.

- **Target systems and users:** We primarily target cloud systems as they experience a wide variety of failures at a higher rate than any other systems in the past [97]. However, our framework is generic for other distributed systems. Our targets so far are HDFS [155], ZooKeeper [107] and Cassandra [120]. We mainly use HDFS as our example in this chapter. In terms of users, we target experienced system developers, with the goal of improving their ability to efficiently generate tests and specifications.

- **Seamless integration:** Our approach requires source code availability. However, for adoptability, our framework should not modify the code base significantly. This is accomplished by leveraging mature interposition technology (*e.g.*, AspectJ). Currently our framework can be integrated to any distributed systems written in Java.

- **Rapid and systematic exploration of failures:** Our framework should help cloud system developers explore multiple-failure scenarios automatically and more systematically than random approaches. However, a complete systematic exploration brings a new challenge: a massive combinatorial explosion of failures, which takes tens of hours to explore. Thus, our testing framework must also be equipped with smart exploration strategies to prioritizing non-similar failure scenarios first.

- **Numerous detailed recovery specifications:** Ideally, developers should be able to write as many detailed specifications as possible. The more specifications written, the finer bug reports produced, the less time needed for debugging. Thus, our framework must meet two requirements. First, the specifications must be developer-friendly (*i.e.*, concise, fast to write, yet easy to understand) or else developers will be reluctant to invest in writing specifications. Second, our framework must facilitate *behavioral* specifications. We note that existing work often focuses on *invariant-like* specifications. This is not adequate as recovery behaves differently under different failure scenarios, and while recovery is still ongoing, the system is likely to go through transient states where some invariants are not satisfied.

In the next subsequent sections, we describe in detail the design of FATE and DESTINI. FATE enables the developers to systematically exercise the system's recovery code in the face of multiple failures. However, after failures are injected, system correctness still requires to be verified. Thus, DESTINI allows the developers to write specifications to verify system correctness in a clear, concise, and precise manner.

3.3 FATE: Failure Testing Service

Within a distributed execution, there are many points in place and time where system components could fail. Thus, our goal is to exercise failures more methodically than random approaches. To achieve this, we present three contributions: a failure abstraction for expressing failure scenarios (Section 3.3.1), a ready-to-use failure service which can be integrated seamlessly to cloud systems (Section 3.3.2), and novel failure prioritization strategies that speed up testing time by an order of magnitude (Section 3.3.3).

3.3.1 Failure IDs: Abstraction For Failures

FATE's ultimate goal is to exercise as many combinations of failures as possible. In a sense, this is similar to model checking which explores different sequences of states. One key technique employed in system model checkers is to record the hashes of the explored states. Similarly in our case, we introduce the concept of *failure IDs*, an abstraction for failure scenarios which can be hashed and recorded in history. A failure ID is composed of an I/O ID and the injected failure (Table 3.2). Below we describe these subcomponents in more detail.

I/O ID Fields			Values
Static	Func. call	:	OutputStream.flush()
	Source File	:	BlockRecv.java (line 45)
Dynamic	Stack trace	:	(the stack trace)
	Node Id	:	Node2
Domain specific	Source	:	Node2
	Dest.	:	Node1
	Net. Mesg.	:	Setup Ack
Failure ID = hash (I/O ID + Crash) = 2849067135			

Table 3.2: **A Failure ID.** *A failure ID comprises an I/O ID plus the injected failure (e.g., crash). Hash is used to record a failure ID.*

- **I/O points:** To construct a failure ID, we choose I/O points (*i.e.*, system/library calls that perform disk or network I/Os) as failure points, mainly for three reasons. First, hardware failures manifest into failed I/Os. Second, from the perspective of a node in distributed systems, I/O points are critical points that either change its internal states or make a change to its outside world (*e.g.*, disks, other nodes). Finally, I/O points are basic operations in distributed systems, and hence an abstraction built on these points can be used for broader purposes.

- **Static and dynamic information:** For each I/O point, the I/O ID is generated from the static (*e.g.*, system call, source file) and dynamic information (*e.g.*, stack trace, node ID) available at the point. Dynamic information is useful to increase failure coverage. For example, recovery might behave differently if a failure happens in different nodes (*e.g.*, first vs. last node in the pipeline).

- **Domain-specific information:** To increase failure coverage further, an I/O ID carries domain-specific information; a common I/O point could write to different file types or send messages to different nodes. FATE's interposition mechanism provides runtime information available at an I/O point such as the target I/O (*e.g.*, file names, IP addresses) and the I/O buffer (*e.g.*, network packet, file buffer). To convert these raw information into a more meaningful context (*e.g.*, "Setup Ack" in Table 3.2), FATE provides an interface that developers can implement. If the interface is empty, FATE can still run, but failure coverage could be sacrificed.

- **Possible failure modes:** Given an I/O ID, FATE generates a list of possible failures that could happen before and after. For example, FATE could throw a bad-disk exception before a disk write, or crash a node after the node receives a message. Currently, we support failures such as crash, permanent disk failure, disk corruption, node-level and rack-level network partitioning, and transient failure. We leave I/O reordering for future work.

3.3.2 Architecture

We built FATE with an aim towards quick and seamless integration to our target systems. Figure 3.2 depicts the four components of FATE: workload driver, failure surface, failure server, and filters.

Workload Driver, Failure Surface, and Server

We first instrument the target system (*e.g.*, HDFS) by inserting a "failure surface". There are many possible layers to insert a failure surface (*e.g.*, inside a system

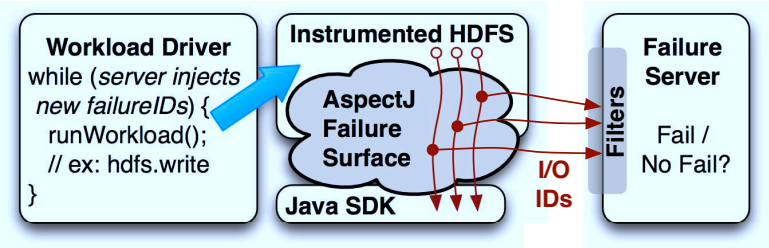

Figure 3.2: **FATE Architecture.** *This figure shows the architecture of* FATE *with four main components: failure surface, workload driver, failure server, and filters. Target systems (e.g., HDFS) are instrumented with a failure surface, which generates failure IDs. The workload driver is where the developers attach the workload to be tested. The failure server systematically explores all possible combinations of failure IDs. Finally, filters control which failures to inject.*

library or at the VMM layer). We do this between the target system and the OS library (*e.g.*, Java SDK), for two reasons. First, at this layer, rich domain-specific information is available. Second, by leveraging mature instrumentation technology (*e.g.*, AspectJ), adding the surface requires no modification to the code base.

The failure surface has two important jobs. First, at each I/O point, it builds the I/O ID. Second, it needs to check if a persistent failure injected in the past affects this I/O point (*e.g.*, network partitioning). If so, the surface returns an error to emulate the failure without the need to talk to the server. Otherwise, it sends the I/O ID to the server and receives a failure decision.

The workload driver is where the developer attaches the workload to be tested (*e.g.*, write, append, or some sequence of operations, including the pre- and post-setups) and specifies the maximum number of failures injected per run. As the workload runs, the failure server receives I/O IDs from the failure surface, combines the I/O IDs with possible failures into failure IDs, and makes failure deci-

sions based on the failure history. The workload driver terminates when the server does not inject a new failure scenario.

Brute-Force Failure Exploration

By default, FATE runs in brute-force mode. That is, FATE systematically explores all possible combinations of observed failure IDs. This is done via *failure locking* and *failure history*. As an example, consider four failure IDs A, B, C, and D, not known a priori. For two-failure scenarios, FATE should exercise AB in one run, AC in another run, and so on. With failure locking, after the first run, the first failure is locked to A (`lock[1]` = A) such that in the next run FATE only injects A for the first failure. For the second failure, since the lock is empty (`lock[2]` = ∅), the server will inject any new failure (*e.g.*, C) as long as the combination (*e.g.*, AC) has not been exercised (in general, for N-failure combinations, FATE only uses `lock[1..N-1]`; `lock[N]` is always empty). If FATE does not observe a new combination that starts with A, the first failure is unlocked and A is recorded in history (`history[1]`={A}) such that in the next run FATE can exercise other combinations that do not start with A (*e.g.*, BC). With this brute-force mode, FATE has exercised over more than 40,000 *unique* combinations of one, two and three failure IDs (*e.g.*, A, BC, and ACD).

Filters

FATE uses information carried in I/O and failure IDs to implement filters at the server side. A filter can be used to regenerate a particular failure scenario or to reduce the failure space. For example, a developer could insert a filter that allows crash-only failures, failures only on some specific I/Os, or failures only at datanodes.

3.3.3 Failure Exploration Strategy

Running FATE in brute-force mode is impractical and time consuming. As an example, we have run the append protocol with a filter that allows crash-only failures on disk I/Os in datanodes. With this filter, injecting two failures per run gives 45 failure IDs to exercise, which leads us to 1199 combinations that take more than 2 hours to run. Without the filter (*i.e.*, including network I/Os and other types of failures) the number will further increase. This introduces the problem of exponential explosion of multiple failures, which has to be addressed given the fact that we are dealing with large code base where an experiment could take more than 5 seconds per run (*e.g.*, due to pre- and post-setup overheads).

Among the 1199 experiments, 116 failed; if recovery is perfect, all experiments should be successful. Debugging all of them led us to 3 bugs as the root causes [76]. Now, we can concretely define the challenge: *Can* FATE *exercise a much smaller number of combinations and find distinct bugs faster?* This section provides some answers to this challenge. To the best of our knowledge, we are the first to address this issue in the context of distributed systems. Thus, we also hope that this challenge attracts system researches to present other alternatives.

To address this challenge, we have studied the properties of multiple failures (for simplicity, we begin with two-failure scenarios). A pair of two failures can be categorized into two types: *pairwise dependent* and *pairwise independent* failures. Below, we describe each category along with the prioritization strategies; we evaluate the algorithms in Section 3.5.3. We also emphasize that our proposed strategies are built on top of the information carried in failure IDs, and hence display the power of the failure IDs abstraction.

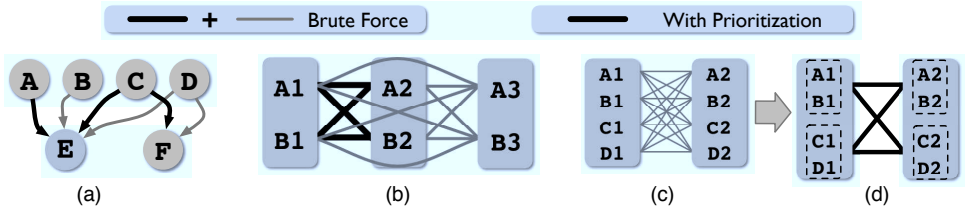

Figure 3.3: **Prioritization of Pairwise Dependent and Independent Failures.** *Notation used in this figure is as follows: letters (e.g., A, B, and C) represent different static failure IDs; letters followed by an index number represent failure IDs that are executed concurrently in different nodes (e.g., A1 and A2 represent the same I/O IDs A that are executed concurrently in two nodes).*

Pairwise Dependent Failures

A pair of failure IDs is dependent if the second ID is *observed* only if the failure on the first ID is *injected*; observing the occurrence of a failure ID does not necessarily mean that the failure must be injected. The key here is to use observed I/Os to capture path coverage information (this is an acceptable assumption since we are dealing with distributed systems where recovery essentially manifests into I/Os). Figure 3.3a illustrates some combinations of dependent failure IDs. For example, `F` is dependent on `C` or `D` (*i.e.*, `F` will never be observed unless `C` or `D` is injected). The brute-force algorithm will inefficiently exercise all six possible combinations: `AE`, `BE`, `CE`, `DE`, `CF`, and `DF`.

To prioritize dependent failure IDs, we introduce *recovery-behavior clustering*, a strategy to prioritize *non-similar* failure scenarios first. The intuition is that non-similar failure scenarios typically lead to different recovery behaviors, and recovery behaviors can be represented as a sequence of failure IDs. Thus, to perform the clustering, we first run a complete set of experiments with *only one* failure per run, and in each run we record the *subsequent* failure IDs.

We formally define subsequent failure IDs as all observed IDs after the injected failure up to the point where the system enters the *stable state*. That is, recording recovery only up to the end of the protocol (*e.g.*, write) is not enough. This is because a failed I/O could leave some "garbage" that is only cleaned up by some background protocols. For example, a failed I/O could leave a block with an old generation timestamp that should be cleaned up by the background replication monitor (outside the scope of the write protocol). Moreover, different failures could leave different types of garbage, and thus lead to different recovery behaviors of the background protocols. By capturing subsequent failure IDs until the stable state, we ensure more fine-grained clustering.

The exact definition of stable state might be different across different systems. For HDFS, our definition of stable state is: FATE reboots dead nodes if any, removes transient failures (*e.g.*, network partitioning), sends commands to the datanodes to report their blocks to the namenode, and waits until all datanodes receive a null command (*i.e.*, no background jobs to run).

Going back to Figure 3.3a, the created mappings between the first failures and their subsequent failure IDs are: {A→ E}, {B→ E}, {C→ E, F}, and {D→ E, F}. The recovery behaviors then are clustered into two: {E}, and {E, F}. Finally, for each recovery cluster, we pick only one failure ID on which the cluster is dependent. The final prioritized combinations are marked with bold edges in Figure 3.3a. That is, FATE only exercises: AE, CE, and CF. Note that E is exercised as a second failure twice because it appears in different recovery clusters.

Pairwise Independent Failures

A pair of failure IDs is independent if the second ID is observed even if the first ID is *not* injected. This case is often observed when the same piece of code runs in parallel, which is a common characteristic found in distributed systems (*e.g.*, two phase commit, leader election, HDFS write and append). Figure 3.3b illustrates a scenario where the same I/O points A and B are executed concurrently in three nodes (*i.e.*, A1, A2, A3, B1, B2, B3). Let's name these two I/O points A and B as static failure points, or SFP in short (as they exclude node ID). With brute-force exploration, FATE produces 24 combinations (the 12 bi-directional edges in Figure 3.3b). In more general, there are $SFP^2 * N(N-1)$ combinations, where N and SFP are the number of nodes and static failure points respectively. To reduce this quadratic growth, we introduce two levels of prioritization: one for reducing $N(N-1)$ and the other for SFP^2.

To reduce $N(N-1)$, we leverage the property of *symmetric code* (*i.e.*, the same code that runs concurrently in different nodes). Because of this property, if a pair of failures has been exercised at two static failure points of two specific nodes, it is not necessary to exercise the same pair for other pairs of nodes. For example, if A1B2 has been exercised, it is not necessary to run A1B3, A2B1, A2B3, and so on. As a result, we have reduced $N(N-1)$ (*i.e.*, any combinations of two nodes) to just one (*i.e.*, a pair of two nodes); the N does not matter anymore.

Although the first level of reduction is significant, FATE still hits the SFP^2 bottleneck as illustrated in Figure 3.3c. Here, instead of having two static failure points, there are four, which lead to 16 combinations. To reduce SFP^2, we utilize the behavior clustering algorithm used in the dependent case. Put simply, the goal is to reduce SFP to $SFP_{clustered}$, which will reduce the input to the quadratic

explosion (*e.g.*, from 4 to 2 resulting in 4 uni-directional edges as depicted in Figure 3.3d). In practice, we have seen a reduction from fifteen SFP to eight $SFP_{clustered}$.

3.3.4 Summary

We have introduced failure IDs as a new abstraction for exploring failures, which we believe is general enough to be used for other purposes (*e.g.*, incorporated to other testing frameworks such as model checkers, to build prioritization policies, *etc.*). Second, we have built a ready-to-use failure service. Deploying FATE is relatively easy; a developer could quickly do that without the domain-specific component. For example, we have ported FATE to two other systems in just a few hours. To increase failure coverage, one can incrementally add the domain-specific fields of failure IDs. Finally, we are the first to present prioritization strategies for exploring multiple failures in distributed systems. Our approaches are not sound; however by experience, all bugs found with brute-force are also found with prioritization (more in Section 3.5.3). If developers have the time and resource, they could fall back to brute-force mode for more confidence.

3.4 DESTINI: Declarative Testing Specifications

After failures are injected, developers still need to verify system correctness. DESTINI attempts to improve the state-of-the-art of writing system specifications. In the following sections, we first describe the architecture (Section 3.4.1), then present some examples (Section 3.4.2), and finally summarize the advantages (Section 3.4.3). Currently, we target recovery bugs that reduce availability (*e.g.*,

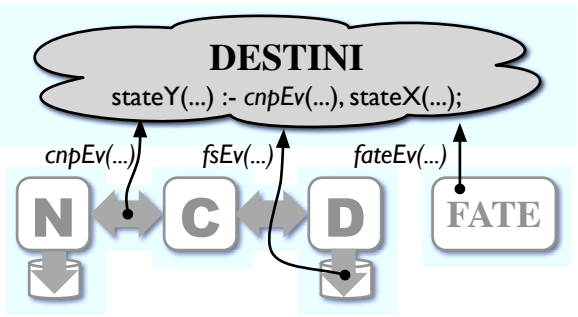

Figure 3.4: **DESTINI Architecture.** DESTINI *interposes network and disk protocols and translates the available information into Datalog events (e.g.,* `cnpEv` *and* `fsEv`*). It also records failure events from* FATE *(e.g.,* `fateEv`*). Finally, based only on events, it records facts, deduces expectations of how the system should behave in the future, and compares the two.*

unmasked failures, fail-stop) and reliability (*e.g.*, data-loss, inconsistency). We leave performance and scalability bugs for future work.

3.4.1 Architecture

At the heart of DESTINI is Datalog, a declarative relational logic language. We chose the Datalog style as it has been successfully used for building distributed systems [19, 126] and for verifying some aspects of system correctness (*e.g.*, security [93, 138]). Unlike much of that work, we are not using Datalog to implement system internals, but only to write correctness specifications that are checked relatively rarely. Hence we are less dependent on the efficiency of current Datalog engines, which are still evolving [19].

In terms of the architecture, DESTINI is designed such that developers can build specifications from minimal information. To support this, DESTINI comprises three features as depicted in Figure 3.4. First, it interposes network and disk protocols and translates the available information into Datalog events (*e.g.*,

cnpEv). Second, it records failure scenarios by having FATE inform DESTINI about failure events (*e.g.*, *fateEv*). This highlights that FATE and DESTINI must work hand in hand, a valuable property that is apparent throughout our examples. Finally, based *only* on events, it records facts, deduces expectations of how the system should behave in the future, and compares the two.

Rule Syntax

In DESTINI, specifications are formally written as Datalog rules. A rule is essentially a logical relation:

```
errX(P1,P2,P3) :- cnpEv(P1), NOT-IN stateY(P1,P2,_),
                  P2 == img, P3 := Util.strLib(P2);
```

This Datalog rule consists of a head table (`errX`) and predicate tables in the body (*cnpEv* and `stateY`). The head is evaluated when the body is true. Tuple variables begin with an upper-case letter (`P1`). A don't care variable is represented with an underscore (_). A comma between predicates represents conjunction. "`:=`" is for assignments. We also provide some helper libraries (`Util.strLib()` to manipulate strings). Lower case variables (`img`) represent integer or string constants. All upper case letters (`NOT-IN`) are Datalog keywords. Events are in italic. To help readers track where events originate from, an event name begins with one of these labels: *cnp*, *dnp*, *cdp*, *ddp*, *fs*, which stand for client-namenode, datanode-namenode, client-datanode, datanode-datanode, and file system protocols respectively (Figure 3.4). Non-event (non-italic) heads and predicates are essentially database tables with primary keys defined in some schemas (not shown). A table that starts with `err` represents an error (*i.e.*, if a specification is broken, the error table is non-empty, implying the existence of one or more bugs).

Data-Transfer Recovery Specifications			
a1	errDataRec (B, N)	:-	*cnpComplete* (B), expectedNodes (B, N), NOT-IN actualNodes (B, N);
a2	pipeNodes (B, Pos, N)	:-	*cnpGetBlkPipe* (UFile, B, Gs, Pos, N);
a3	expectedNodes (B, N)	:-	pipeNodes (B, Pos, N);
a4	DEL expectedNodes (B, N)	:-	*fateCrashNode* (N), pipeStage (B, Stg), Stg == 2, expectedNodes (B, N);
a5	setupAcks (B, Pos, Ack)	:-	*cdpSetupAck* (B, Pos, Ack);
a6	goodAcksCnt (B, COUNT<Ack>)	:-	setupAcks (B, Pos, Ack), Ack == 'OK';
a7	nodesCnt (B, COUNT<Node>)	:-	pipeNodes (B, _, N, _);
a8	pipeStage (B, Stg)	:-	nodesCnt (NCnt), goodAcksCnt (ACnt), NCnt == Acnt, Stg := 2;
a9	blkGenStamp (B, Gs)	:-	*dnpNextGenStamp* (B, Gs);
a10	blkGenStamp (B, Gs)	:-	*cnpGetBlkPipe* (UFile, B, Gs, _, _);
a11	diskFiles (N, File)	:-	*fsCreate* (N, File);
a12	diskFiles (N, Dst)	:-	*fsRename* (N, Src, Dst), diskFiles (N, Src, Type);
a13	DEL diskFiles (N, Src)	:-	*fsRename* (N, Src, Dst), diskFiles (N, Src, Type);
a14	fileTypes (N, File, Type)	:-	diskFiles(N, File), Type := Util.getType(File);
a15	blkMetas (N, B, Gs)	:-	fileTypes (N, File, Type), Type == metafile, B := Util.getBlk(File), Gs := Util.getGs(File);
a16	actualNodes (B, N)	:-	blkMetas (N, B, Gs), blkGenStamp (B, Gs);

Table 3.3: **Sample Specifications.** *Together with table 3.4, this table lists rules that we wrote to specify the problems in Section 3.4.2. All logical relations are built only from events (in italic). The shaded rows indicate checks that catch violations. A check always starts with* `err`*. Tuple variables* `B`*,* `Gs`*,* `N`*,* `Pos`*,* `R`*,* `Stg`*,* `NnFile`*, and* `UFile` *are abbreviations for block, generation timestamp, node, position, rack, stage, namenode file, and user file respectively; others should be self-explanatory. Each table has primary keys defined in a schema (not shown).*

Tighter Specifications for Data-Transfer Recovery			
b1	errBadAck (Pos, N)	:-	*cdpDataAck* (Pos, 'Error'), pipeNodes (B, Pos, N), liveNodes (N);
b2	liveNodes (N)	:-	*dnpRegistration* (N);
b3	DEL liveNodes (N)	:-	*fateCrashNode* (N);
b4	errBadConnect (N, TgtN)	:-	*ddpDataTransfer* (N, TgtN, Status), liveNodes (TgtN), Status == terminated;

Rack-Aware Policy Specifications			
c1	warnSingleRack (B)	:-	rackCnt (B, 1), actualRacks (B, R), connectedRacks (R, OtherR);
c2	actualRacks (B, R)	:-	actualNodes (B, N), nodeRackMap (N, R);
c3	rackCnt (B, COUNT<R>)	:-	actualRacks (B, R);
c4	DEL connectedRacks (R1, R2)	:-	fatePartitionRacks (R1, R2);
c5	err1RackOnCompletion (B)	:-	*cnpComplete* (B), warnSingleRack (B);
c6	err1RackOnStableState (B)	:-	*fateStableState* (_), warnSingleRack (B);

Refining Log-Recovery Specifications			
d1	errLostUFile (UFile)	:-	expectedUFile (UFile), NOT-IN ufileInNameNode (UFile);
d2	ufileInNameNode (UFile) **	:-	ufileInNnFile(F, NnFile), (NnFile == img || NnFile == log || NnFile == img2);
d3	ufileInNameNode (UFile)	:-	ufileInNnFile (F, img2), logRecStage (Stg), Stg == 4;
d4	ufileInNameNode (UFile)	:-	ufileInNnFile (F, img) , logRecStage (Stg), Stg != 4;
d5	ufileInNameNode (UFile)	:-	ufileInNnFile (F, log) , logRecStage (Stg), Stg != 4;

Table 3.4: **Sample Specifications (continued).** *(**) Rule* d2 *is refined in* d3 *to* d5.

3.4.2 DESTINI Examples

This section presents the powerful features of DESTINI via four examples of HDFS recovery specifications. In the first example, we present five important components of recovery specifications. To help the complex debugging process, the second example shows how developers can incrementally add tighter specifications. The third example presents specifications that incorporate a different type of failure than the first two examples. Finally, we illustrate how developers can refine existing specifications.

Specifying Data-Transfer Recovery

DESTINI facilitates five important elements of recovery specifications: checks, expectations, facts, precise failure events, and check timings. Here, we present these elements by specifying the data-transfer recovery protocol (Figure 3.1a); this recovery is correct if valid replicas are stored in the surviving nodes of the pipeline.

- **Checks:** To catch violations of data-transfer recovery, we start with a simple high-level *check* (**a1**), which says "upon block completion, throw an error if there is a node that is expected to store a valid replica, but actually does not." This rule shows how a check is composed of three elements: the *expectation* (`expectedNodes`), *fact* (`actualNodes`), and *check timing* (*`cnpComplete`*).

- **Expectations:** The expectation (`expectedNodes`) is deduced from protocol events (**a2**-**a8**). First, without any failure, the expectation is to have the replicas in all the nodes in the pipeline (**a3**); information about pipeline

nodes are accessible from the setup reply from the namenode to the client (**a2**). However, if there is a crash, the expectation changes: the crashed node should be removed from the expected nodes (**a4**). This implies that an expectation is also based on *failure events*.

- **Failure events:** Failures in different stages result in different recovery behaviors. Thus, we must know precisely when failures occur. For data-transfer recovery, we need to capture the current stage of the write process and only change the expectation if a crash occurs within the data-transfer stage (*fateCrashNode* happens at `Stg==2` in rule **a4**). The data transfer stage is deduced in rules **a5**-**a8**: the second stage begins after all acks from the setup phase have been received.

  Before moving on, we emphasize two important observations here. First, this example shows how FATE and DESTINI must work hand in hand. That is, recovery specifications require a failure service to exercise them, and a failure service requires specifications of expected failure handling. Second, with logic programming, developers can easily build expectations only from events.

- **Facts:** The fact (`actualNodes`) is also built from events (**a9**-**a16**), more specifically, by tracking the locations of valid replicas. A valid replica can be tracked with two pieces of information: the block's latest generation time stamp, which DESTINI tracks by interposing two interfaces (**a9** and **a10**), and meta/checksum files with the latest generation timestamp, which are obtainable from file operations (**a11**-**a15**). With this information, we build the runtime fact: the nodes that store the valid replicas of the block (**a16**).

- **Check timings:** The final step is to compare the expectation and the fact. We underline that the timing of the check is important because we are specifying *recovery behaviors*, unlike invariants which must be true at all time. Not paying attention to this will result in false warnings (*i.e.*, there is a period of time when recovery is ongoing and specifications are not met). Thus, we need precise events to signal check times. In this example, the check time is at block completion (`cnpComplete` in **a1**).

Debugging with Tighter Specifications

The rules in the previous section capture the high-level objective of HDFS data-transfer recovery. After we ran FATE to cover the first crash scenario in Figure 3.1b (for simplicity of explanation, we exclude the second crash), rule **a1** throws an error due to a bug that wrongly excludes the good second node (Figure 3.1b in Section 3.1.3). Although, the check unearths the bug, it does not *pinpoint* the bug (*i.e.*, answer *why* the violation is thrown).

To help this debugging process, we added more detailed specifications. In particular, from the events that DESTINI logs, we observed that the client excludes the second node in the next pipeline, which is possible if the client receives a bad ack. Thus, we wrote another check (**b1**) which says "throw an error if the client receives a bad ack for a live node" (**b1**'s predicates are specified in **b2** and **b3**). Note that this check is written from the *client's view*, while rule **a1** is written from the *global view*.

The new check catches the bug closer to the source, but also raises a new question: Why does the client receive a bad ack for the second node? One logical explanation is because the first node cannot communicate to the second node.

Thus, we easily added many checks that catch unexpected bad connections such as **b4**, which finally pinpoints the bug: the second node, upon seeing a failed connection to the crashed third node, incorrectly closes the streams connected to the first node; note that this check is written from the *datanode's view*.

In summary, more detailed specifications prove to be valuable for assisting developers with complex debugging process. This is unlikely to happen if a check implementation is long. But with DESTINI, a check can be expressed naturally in a small number of logical relations. Moreover, checks can be written from different views (*e.g.*, global, client and datanode as shown in **a1**, **b1**, **b4** respectively). Table 3.5 shows a timeline of when these checks are violated. As shown, tighter specifications essentially fill the "explanation gaps" between the injected failure and the wrong final state of the system.

Specifying Rack-Aware Replication Policy

In this example, we write specifications for the HDFS rack-aware replication policy, an important policy for high availability [87, 155]. Unlike previous examples, this example incorporates the network partitioning failure mode. According to the HDFS architects [155], the write protocol should ensure that block replicas are spread across a minimum of two available racks. But, if only one rack is reachable, it is acceptable to use one rack temporarily. To express this, rule **c1** throws a warning if a block's rack could reach another rack, but the block's rack count is one (rules **c2-c4** provide topology information, which is initialized when the cluster starts and updated when FATE creates a rack partition). This warning becomes a hard error *only* if it is true upon block completion (**c5**) or stable state (**c6**). Note again how these timings are important to prevent false errors; while recovery is ongoing, replicas are still being re-shuffled into multiple racks.

Time, Events, and Errors

t1: Client asks the namenode for a block ID and the nodes.
`cnpGetBlkPipe (usrFile, blk_x, gs1, 1, N1);`
`cnpGetBlkPipe (usrFile, blk_x, gs1, 2, N2);`
`cnpGetBlkPipe (usrFile, blk_x, gs1, 3, N3);`

t2: Setup stage begins (pipeline nodes setup the files). *
`fsCreate (N1, tmp/blk_x_gs1.meta);`
`fsCreate (N2, tmp/blk_x_gs1.meta);`
`fsCreate (N3, tmp/blk_x_gs1.meta);`

t3: Client receives setup acks. Data transfer begins.
`cdpSetupAck (blk_x, 1, OK);`
`cdpSetupAck (blk_x, 2, OK);`
`cdpSetupAck (blk_x, 3, OK);`

t4: FATE crashes N3. **Got error (b4).**
`fateCrashNode (N3);`
`errBadConnect (N1, N2); // should be good`

t5: Client receives an erroneous ack. **Got error (b1).**
`cdpDataAck (2, Error);`
`errBadAck (2, N2); // should be good`

t6: Recovery begins. Get new generation time stamp. `dnpNextGenStamp (blk_x, gs2);`

t7: Only N1 continues and finalizes the files.
`fsCreate (N1, tmp/blk_x_gs2.meta);`
`fsRename (N1, tmp/blk_x_gs2.meta,`
`current/blk_x_gs2.meta);`

t8: Client marks completion. **Got error (a1).**
`cnpComplete (blk_x);`
`errDataRec (blk_x, N2); // should exist`

Table 3.5: **A Timeline of DESTINI Execution.** *The table shows the timeline of runtime events (italic) and errors (shaded). Tighter specifications capture the bug earlier in time. The tuples (strings/integers) are real entries (not variable names). For space, we do not show block-file creations (but only meta files***) *nor how the rules in Tables 3.3 and 3.4 are populated.*

With these checks, DESTINI found the bug in Figure 3.1c (Section 3.1.3), a critical bug that could greatly reduce availability: all replicas of a block are stored in a single rack. More specifically, the bug does not violate the completion rule (because the racks are still partitioned). But, it does violate the stable state rule because even after the network partitioning is removed, the replication monitor does not re-shuffle the replicas.

Refining Specifications

The developers can not only *incrementally add* detailed specifications but also *refine* existing specifications. Here, we specify the HDFS log-recovery process in order to catch data-loss bugs in this protocol. The high-level check (**d1**) is fairly simple: "a user file is lost if it does not exist at the namenode." To capture the facts, we wrote rule **d2** which says "*at any time*, user files should exist in the union of all the three namenode files used in log recovery." With these rules, we found a data-loss bug that accidentally deletes the metadata of user files. But, the error is only thrown *at the end* of the log recovery process (*i.e.*, the rules are not detailed enough to pinpoint the bug). We then refined rule **d2** to reflect in detail the four stages of the process (**d3** to **d5**). That is, depending on the stage, user files are expected to be in a different subset of the three files. With these refined specifications, the data-loss bug was captured in between stage 3 and 4.

3.4.3 Summary of Advantages

Throughout the examples, we have shown the advantages of DESTINI: it facilitates checks, expectations, facts, failure events, and precise timings; specifications can be written from different views (*e.g.*, global, client, datanode); different

types of violations can be specified (*e.g.*, availability, data-loss); different types of failures can be incorporated (*e.g.*, crashes, partitioning); and specifications can be incrementally added or refined. Overall, the resulting specifications are clear, concise, and precise, which potentially attracts developers to write many specifications to ease complex debugging process, for both present and future related bugs. All of these are feasible due to three important properties of DESTINI: the interposition mechanism that translates disk and network events; the use of relational logic language which enables us to deduce complex states only from events; and the inclusion of failure events from the collaboration with FATE.

3.5 Evaluation

We evaluate FATE and DESTINI in several aspects: the general usability for cloud systems (Section 3.5.1), the ability to catch multiple-failure bugs (Section 3.5.2), the efficiency of our prioritization strategies (Section 3.5.3), the number of specifications we have written and their reusability (Section 3.5.4), the number of new bugs we have found and old bugs reproduced (Section 3.5.5), and the implementation complexity (Section 3.5.6).

3.5.1 Target Systems and Protocols

We have integrated FATE and DESTINI into three cloud systems: HDFS v0.20.0 and v0.20.2+320 (the latter is released in Feb. 2010 and used by Cloudera and Facebook), ZooKeeper v3.2.2 (Dec. 2009), and Cassandra v0.6.1 (Apr. 2010). We have run our framework on four HDFS workloads (log recovery, write, append, and replication monitor), one ZooKeeper workload (leader election), and one Cassandra workload (key-value insert).

3.5.2 Multiple-Failure Bugs

The uniqueness of our framework is the ability to explore multiple failures systematically, and thus catch corner-case multiple-failure bugs. Here, we describe two out of five multiple-failure bugs that we found [76].

Append Bugs

We begin with a multiple-failure bug in the HDFS append protocol. Unlike write, append is more complex because it must atomically mutate block replicas [178]. HDFS developers implement append with a custom protocol; their latest append design was written in a 19-page document of prose specifications [117]. Append was finally supported after being a top user demand for three years [178]. As a note, GFS also supports append, but its authors did not share their internal design [87].

The experiment setup was that a block has three replicas in three nodes, and thus should survive two failures. On append, the three nodes form a pipeline. N1 starts a thread that streams the new bytes to N2 and then N1 appends the bytes to its block. N2 crashes at this point, and N1 sends a bad ack to the client, but does not stop the thread. Before the client continues streaming via a new pipeline, all surviving nodes (N1 and N3) must agree on the same block offset (the `syncOffset` process). In this process, each node stops the writing thread, verifies that the block's in-memory and on-disk lengths are the same, broadcasts the offset, and picks the smallest offset. However, N1 might have not updated the block's in-memory length, and thus throws an exception resulting in the new pipeline containing only N3. Then, N3 crashes, and the pipeline is empty. The append fails, but worse, the block in N1 (still alive) becomes "trapped" (*i.e.*, inaccessible). After FATE ran all the background protocols (*e.g.*, lease recovery), the block is still

trapped and permanently inaccessible. We have submitted a fix for this bug [10].

Combinations of Different Failures

We have also found a new data-loss bug due to a sequence of *different* failure modes, more specifically, transient disk failure (#1), crash (#2), and disk corruption (#3) at the namenode. The experiment setup was that the namenode has three replicas of metadata files on three disks, and one disk is flaky (exhibits transient failures and corruptions). When users store new files, the namenode logs them to all the disks. If a disk (*e.g.*, Disk1) returns a transient write error (#1), the namenode will exclude this disk; future writes will be logged to the other two disks (*i.e.*, Disk1 will contain stale data). Then, the namenode crashes after several updates (#2). When the namenode reboots, it will load metadata from the disk that has the latest update time. Unfortunately, the file that carries this information is not protected by a checksum. Thus, if this file is corrupted (#3) such that the update time of Disk1 becomes more recent than the other two, then the namenode will load stale data, and flush the stale data to the other two disks, wiping out all recent updates. One could argue that this case is rare, but cloud-scale deployments cause rare bugs to surface; a similar case of corruption did occur in practice [10]. Moreover, data-loss bugs are serious ones [128, 131, 135].

3.5.3 Prioritization Efficiency

When FATE was first deployed without prioritization, we exercised over 40,000 unique combinations of failures, which combine into 80-hour of testing time. Thousands of experiments failed (probably only due to tens of bugs). This was an overwhelming situation which fortunately unfolded into a good outcome: new

Workload	#Failure	Strategy	#EXP	FAIL	BUGS
Append	2	Brute-force	**1199**	116	**3**
		Prioritization	**112**	17	**3**
Append	3	Brute-force	**7720**	**3693	*3
		Prioritization	**618**	72	*3
Write	2	Brute-force	**524**	120	**2**
		Prioritization	**49**	27	**2**
Write	3	Brute-force	**3221**	911	*2
		Prioritization	**333**	82	*2

Table 3.6: **Prioritization Efficiency.** *The columns from left to right are the number of injected failures per run, exploration strategy, combinations/experiments (EXP), failed experiments (FAIL), and bugs found (BUGS). Note that the bug counts are only due to two and three failures and depend on the filter (i.e., there are more bugs than shown). Each experiment takes 4 seconds on average. (*) Bugs in three-failure experiments are the same as in two-failure ones. (**) This high number is due to a design bug; we used triaging to help us classify the bugs.*

strategies for multiple-failure prioritization.

To evaluate our strategies, we first focused only on two protocols (write and append) because we need to compare the brute-force with the prioritization results. More specifically, for each method, we count the number of combinations and the number of distinct bugs. Our hope is that the latter is the same for brute-force and prioritization. Table 3.6 shows the result of running the two workloads with two and three failures per run, and with a lightweight filter (crash-only failures on disk I/Os in datanodes); without this filter, the number of brute-force experiments is too large to debug. In short, the table shows that our prioritization strategies reduce the total number of experiments by an order of magnitude, and from our experience no bugs are missing. Again, we cannot prove that our approach is sound; developers could fall back to brute-force for more confidence.

3.5.4 Specifications

In the last six months, we have written 74 checks on top of 174 rules for a total of 351 lines of Datalog code (65 checks for HDFS, 2 for ZooKeeper, and 7 for Cassandra). We want to emphasize that $\frac{rules}{checks}$ ratio displays how DESTINI empowers specification reuse (*i.e.*, building more checks on top of existing rules). As a comparison, the ratio for our first check (Section 3.4.2 in Table 3.3) is 16:1, but the ratio now is 3:1.

Table 3.7 compares DESTINI with other related works on testing frameworks that empower system specifications such as D3S [124], Pip [146], WiDS [125], and P2 Monitor [157]. These works support specifications written in either declarative or scripting languages. For instance, D3S [124] and WiDS [125] employ a scripting language that still requires a check to be written in tens of lines of code; Pip [146] facilitates declarative checks, but a check is still written in over 40 lines on average. The table highlights that DESTINI allows a large number of checks to be written in smaller lines of code. We want to note that the number of specifications we have written so far only represents six recovery protocols; there are more that can be specified. As time progresses, we believe the simplicity offered by DESTINI will open the possibility of having hundreds of specifications along with more recovery specification patterns.

To show how our style of writing specifications is applicable to other systems, we present in more detail some specifications we wrote for ZooKeeper and Cassandra.

Spec. Language	Framework	#Checks	Lines/Check
Scripting	D3S [124]	10	**53**
Declarative	Pip [146]	44	**43**
Scripting	WiDS [125]	15	**22**
Declarative	P2 Monitor [157]	11	**12**
Declarative	DESTINI	74	**5**

Table 3.7: **DESTINI vs. Related Work.** *The table compares* DESTINI *with related works on testing frameworks that empower system specifications in terms of specification languages, (e.g., declarative and scripting), the number of checks written, and the average lines of code per check.*

ZooKeeper

We have integrated our framework to ZooKeeper [107]. We picked two reported bugs in the version we analyzed. Let's say three nodes N1, N2, and N3, participate in a leader election, and $id(N1) < id(N2) < id(N3)$. If N3 crashes at any point in this process, the expected behavior is to have N1 and N2 form a 2-quorum. However, there is a bug that does not anticipate N3 crashing at a particular point, which causes N1 and N2 to continue nominating N3 in ever-increasing rounds. As a result, the election process never terminates and the cluster never becomes available. To catch this bug, we wrote an invariant violation "a node chooses a winner of a round without ensuring that the chosen leader has in itself voted in the round." The other bug involves multiple failures and can be caught with an addition of just one check; we reuse rules from the first bug. So far, we have written 12 rules for ZooKeeper.

Cassandra

We have also done the same for Cassandra [120], and picked three reported bugs in the version we analyzed. In Cassandra, the key-value insert protocol allows users to specify a consistency level such as `one`, `quorum`, or `all`, which ensures that the client waits until the key-value has been flushed on at least one, N/2 + 1, or all N nodes respectively. These are simple specifications, but again, due to complex implementation, bugs exist and break the rules. For example, at level `all`, Cassandra could incorrectly return a success even when only one replica has been completed. FATE is able to reproduce the failure scenarios and DESTINI is equipped with 7 checks (in 12 rules) to catch consistency-level related bugs.

3.5.5 New Bugs and Old Bugs Reproduced

We tested HDFS for over eight months and submitted 16 new bugs [76], out of which, 7 led to design bugs (*i.e.*, require protocol modifications) and 9 led to implementation bugs. All have been confirmed by the developers. We tested Cassandra and ZooKeeper for only two months, observed some failed experiments, but since we did not have the chance to debug all of them, we had no new bugs to report.

To further show the power of our framework, we address two challenges: Can FATE reproduce all the failure scenarios of old bugs? Can DESTINI facilitate specifications that catch the bugs? The idea is that before proposing our framework for catching unknown bugs, we wanted to feel confident that it is expressive enough to capture known bugs. We went through the 91 HDFS recovery issues (Section 3.1.2) and selected 74 that relate to our target workloads (Section 3.5.1).

FATE is able to reproduce all of them; as a proof, we have created 22 filters (155 lines in Java) to reproduce all the scenarios. Furthermore, we have written checks that could catch 46 old bugs; since some of the old bugs have been fixed in the version we analyzed, we introduced artificial bugs to test our specifications. For ZooKeeper and Cassandra, we have reproduced a total of five bugs.

3.5.6 FATE and DESTINI Complexity

FATE comprises generic (workload driver, failure server, failure surface) and domain-specific parts (workload driver, I/O IDs). The generic part is written in 3166 lines in Java. The domain-specific parts are 422, 253, and 357 lines for HDFS, ZooKeeper and Cassandra respectively; the part for HDFS is bigger because HDFS was our first target. DESTINI's implementation cost comes from the translation mechanism (Section 3.4.1). The generic part is 506 lines. The domain-specific parts are 732 (more complete), 23, and 35 lines for HDFS, ZooKeeper, and Cassandra respectively. FATE and DESTINI interpose the target systems with AspectJ (no modification to the code base). However, it was necessary to slightly modify the systems (less than 100 lines) for two purposes: deferring background tasks while the workload is running and sending stable-state commands.

3.6 Conclusion

The scale of cloud systems – in terms of both infrastructure and workload – makes failure handling an urgent challenge for system developers. To assist developers in addressing this challenge, we have presented FATE and DESTINI as a new framework for cloud recovery testing. We believe that developers need both FATE and

DESTINI as a unified framework: recovery specifications require a failure service to exercise them, and a failure service requires specifications of expected failure handling. Overall, we have presented five specific contributions:

- A ready-to-use testing framework that exercises multiple failures systematically via the use of a new failure abstraction (failure IDs).
- The first prioritization strategies for exploring multiple failures in distributed systems, which explore distinct recovery behaviors an order of magnitude faster than a brute-force approach.
- A framework for writing specifications in a relational logic language, which enables developers to write clear and concise recovery specifications.
- Design patterns for writing recovery specifications (*e.g.*, how to capture facts, build expectations, specify check timings, express different types of violations, incorporate different types of failures, *etc.*).
- The results of applying our framework to three widely-used cloud systems (HDFS, ZooKeeper, and Cassandra).

Beyond finding problems in existing systems, we believe such testing is also useful in helping to generate new ideas on how to build robust, recoverable systems. Only through further careful testing and analysis will the next generation of cloud systems meet their demands.

Chapter 4

Handling Fail-silent Failures with SLEEVE

Modern software systems must deal with memory corruption and software bugs that are becoming more common. Therefore, we address a failure model where in-memory data can contain wrong values due to memory corruption and software bugs. If not handled properly, these errors lead to fail-silent behaviors that are hard to detect and can cause severe problems like data loss and service unavailability. We assume that the system is not malicious and that persistent storage is trusted.

Figure 4.1 illustrates some problems caused by fail-silent behaviors. Figure 4.1a shows a normal correct behavior of HDFS; a client writes a file `F` and the HDFS namenode replicates `F`'s data block, `D`, to two datanodes (in 2-way replication). However, silent memory corruption such as a bit flip can take place (*e.g.*, metadata `F` flips to `G` in Figure 4.1b). In this case, the user will not be able to read the file in the future. Subtle software bugs in HDFS (Section 4.4.1) could also lead to silent data loss or corruption. For example, in Figure 4.1c, a bug in the namenode silently deletes `F`'s data blocks in a background task.

In this chapter, we attempt to address this question: How should distributed storage systems deal with fail-silent behaviors efficiently? Many approaches such as Byzantine fault tolerance (BFT) [121], N-version programming [29, 32], and the use of ECC memory have been proposed. However, existing approaches either incur high performance overhead, hardware cost, or engineering effort. Thus, we propose a new approach: selective and lightweight versioning (SLEEVE). The SLEEVE approach advocates that developers selectively protect important subsets of the target system in a lightweight and approximate manner. We apply the concept of SLEEVE to harden three important components of HDFS to form HARDFS. We show that HARDFS detects and recovers from a wide range of fail-silent behaviors, and incurs small performance and space overheads.

In the rest of this chapter, we describe the SLEEVE approach and present the HARDFS case study. We first discuss the SLEEVE approach and its usefulness (Section 4.1), then describe the HARDFS design (Section 4.2), our implementation (Section 4.3) and evaluation (Section 4.4).

4.1 The SLEEVE Approach

The goal of the SLEEVE approach is to selectively protect some part of the target system against fail-silent behaviors and to do so in a lightweight manner (with little space and performance overhead). Figure 4.1d illustrates HARDFS, an HDFS system that employs the SLEEVE approach. The code of the HDFS system (which we call the *main version*) implements the complete functionality of the system. A developer can pick some important piece of functionality and create a "second version" of it, a variant of 2-version programming. This second, selective, and

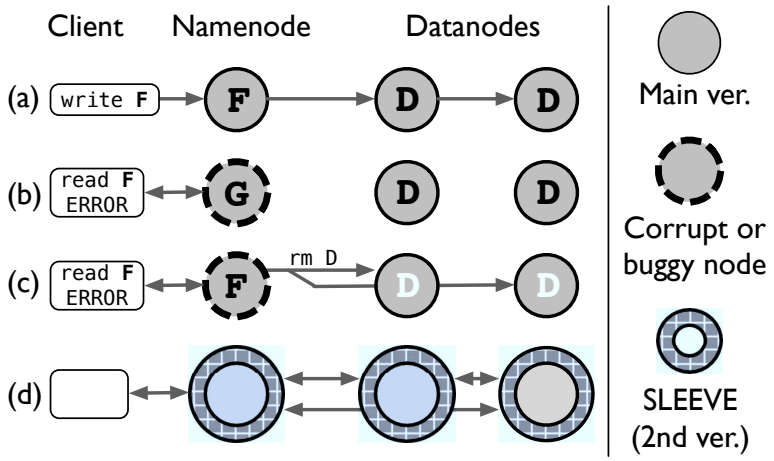

Figure 4.1: **HDFS, corrupted HDFS, and HARDFS.** *This figure illustrates some problems caused by fail-silent behaviors in HDFS and the usefulness of HARDFS. Figure (a) shows a normal correct behavior of HDFS; a client writes a file* `F` *and the HDFS namenode replicates* `F`*'s data block,* `D`*, to two datanodes (in 2-way replication). Figure (b) shows a case of silent memory corruption (e.g., metadata* `F` *flips to* `G`*). In Figure (c), a bug in the namenode silently deletes* `F`*'s data blocks in a background task. Figure (d) shows an example of a sleeved system (HARDFS); sleeved layers watch inputs and outputs of the main version to detect incorrect behaviors.*

lightweight version models the state and logic of the main version. The model can detect misbehavior in the main version and trigger appropriate responses. We refer to systems that pair a complete main version with a modeled second version as *sleeved systems*.

Sleeved systems watch inputs and outputs of the main version (as illustrated in Figure 4.1d) to detect incorrect behaviors that deviate from the model. For example, memory corruption and software bugs in Figure 4.1b and 4.1c can easily be detected; HARDFS will catch the read `F` error and incorrect background data removal (`rm D`) as faulty behaviors. After detecting faulty behaviors, a sleeved system can perform an appropriate action, such as micro-recovery, to transform

faulty states (*e.g.*, corrupt metadata in the main version memory) into consistent states. Thus, a sleeved system isolates faulty behavior within a single node; faults are not propagated to persistent storage or other nodes.

We have three requirements for hardening HDFS which the SLEEVE approach satisfies: HARDFS should be effective at detecting and handling faults (Section 4.4.1), the additional protection should incur minimal performance and memory overhead (Section 4.4.2), and hardening HDFS should require reasonable engineering effort (Section 4.4.3).

The first and second requirements are satisfied because SLEEVE is selective (Section 4.1.1) and lightweight (Section 4.1.2). The third requirement is satisfied in our evaluation (Section 4.4).

4.1.1 Selective Versioning

While traditional N-versioning requires developers to re-implement *all* the functionality of the specification, *selective versioning* requires an additional version for only the most important functionality. The idea is that some functionality in the system is worth protecting more than other functionality, for several reasons.

First, some components are more *sensitive* to bugs and memory corruption. For instance, a bug in the HDFS namespace or replica management could cause irrecoverable data loss; a buggy transaction committed to the log can make the system crash permanently; corrupt internal state could make the system serve incorrect data. On the other hand, bugs in maintaining system statistics may be less harmful. Therefore, if one must prioritize, it is more appropriate to protect bug-sensitive functionalities first.

Other potential candidates for applying the SLEEVE approach are modules that

are *new* or *frequently changed*. Real-world cases have shown that code that does not change frequently is relatively stable, and hence less likely to contain bugs, while new or frequently-changed code is more likely to be buggy [92, 160, 186].

Finally, some software systems already contain protection machinery for some modules. For example, HDFS on-disk data is already protected with checksums and replication [141, 179], and thus the second version could just protect the exposed in-memory system metadata. HARDFS hardens namespace management, replica management, and the read/write protocol of HDFS.

4.1.2 Lightweight Versioning

With *lightweight versioning*, we avoid completely replicating the state maintained by the main version. This challenge particularly arises when a single node needs to store a large amount of state. For example, the HDFS namespace management could manage in-memory metadata of millions of files in one machine.

A naive approach for a 2-version system is to maintain the same amount of metadata in the second version as the main version. Although simple, this approach is unattractive because of its large memory overhead (potentially 100%). When memory is scarce, this design choice limits system scalability. For instance, doubling memory overhead could reduce the maximum number of files the system can manage [156]. Moreover, many systems may run on the same cluster (*e.g.*, Hadoop MapReduce, HBase, and HDFS), so doubling memory overhead is undesirable.

We exploit compact encoding techniques to minimize memory overheads. We have found that sleeved systems can be organized to ask boolean questions (*e.g.*, "Does file `F` really exist?"); therefore, we can use efficient encodings that answer

boolean questions. HARDFS uses a Bloom filter to efficiently encode the file hierarchy for our sleeved namespace management functionality which could contain millions of files. We describe our lightweight approach more in Section 4.2.3

4.1.3 Recovery

Detecting faults that are normally silent is the primary contribution of SLEEVE. Upon detection, a variety of standard recovery techniques or tools can be used, such as: restart, *fsck*, safemode or otherwise blocking dangerous actions, or failover.

In addition to simply detecting errors, SLEEVE can often pinpoint the problem, enabling sophisticated recovery options, such as *micro-recovery*, a fast alternative to full reboot. Fail-silent behaviors sometimes occur due to state corruption; with a second version of the internal state, the system can pinpoint and correct only the corrupt state. With the available redundancy, a sleeved system can initiate fast, fine-grained recovery as opposed to slow, coarse-grained recovery. In HARDFS, we use multiple techniques. When the main version is about to respond to a request with bad data, we try micro-recovery first and fall back on full reboot if necessary. When a node sends requests that appear misguided and could result in data loss, HARDFS blocks the action. We describe the recovery of HARDFS in detail in Section 4.2.5.

4.1.4 Soundness and Completeness

SLEEVE is not sound: we do not attempt to guarantee that a sleeved system never triggers recovery action unnecessarily. Like the main version, the second version is also subject to anomalous bit flips and bugs. As long as recovery actions have a

small cost, occasional false positives are acceptable. SLEEVE is not complete: we do not attempt to catch all faults. Our premise is that faults are more dangerous in some subsystems than others, and complete checking is not possible without a formal specification of behavior regardless. Although SLEEVE fault detection is neither sound nor complete, our experiments show that it is quite useful for handling memory corruption and real software bugs (Section 4.4).

4.2 HARDFS Design

We now describe our general approach to designing HARDFS with SLEEVE. In Section 4.3, we describe in detail how we implement the design to harden the HDFS namespace management, replica management, and the read/write protocol of datanodes with HARDFS-N, HARDFS-R, and HARDFS-D respectively.

4.2.1 Node Models

HDFS nodes can be described by a *behavioral model*: nodes perform *actions* in response to *events*. Events occur when a node receives *input* messages from other systems or when *periodic threads* trigger work. The actions a node performs include modifying the node's *memory state*, accessing persistent storage, and sending output messages based on the current state. In HARDFS, sleeved subsystems understand the behavior model. A node is considered faulty if it performs *incorrect actions* (*i.e.*, actions that deviate from the model).

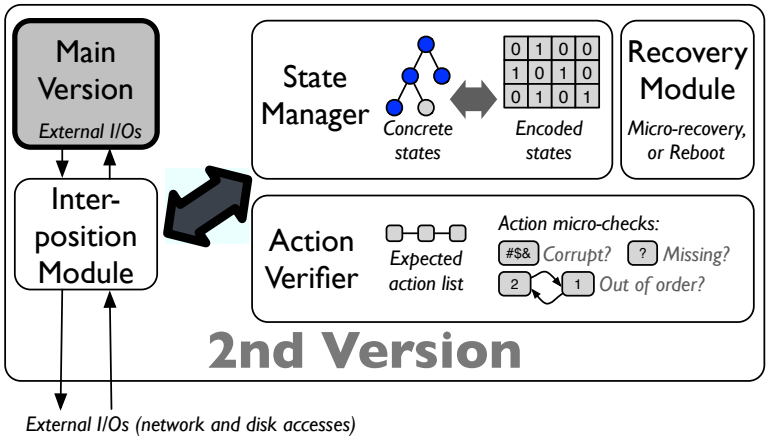

Figure 4.2: **Sleeved systems architecture.** *Each sleeved subsystem has four major modules: an interposition module to interpret semantics of external I/Os and thread events, a state manager to maintain bookkeeping information, an action verifier to detect faulty actions, and a recovery module to perform proper recovery such as micro-recovery and crash and reboot.*

4.2.2 Hardened Subsystem Architecture

To harden a distributed storage system against incorrect actions, we augment each node in the system with a lightweight version that verifies node behavior. More specifically, we "sleeve" each node by interposing on message and file I/O without significantly changing the core implementation. With this approach, faulty behaviors are also isolated within a single node and not propagated to persistent storage or other nodes. As depicted in Figure 4.2, we use four major modules for each sleeved subsystem: an *interposition module*, a *state manager*, an *action verifier*, and a *recovery module*.

- **Interposition module:** A sleeved system forwards all input messages to the main version, and forwards the messages relevant to the hardened functionality to the state manager. It also interposes on thread events to know when

a periodic thread is triggered. This interposition is important because a periodic thread may trigger events that change the state of the main version; the second version must make equivalent changes to its own model. HARDFS uses AspectJ [5] to interpose on events without making major changes to the main version. Due to its straight-forward nature, we do not describe interposition further.

- **State manager:** The state-manager module of HARDFS does the bookkeeping necessary to describe and check the data maintained in the main version. To be lightweight, the state manager keeps the state of the hardened functionality in *encoded states*. HARDFS encodes states with Bloom filters, but a variety of data structures could be used for this purpose. Since encoding techniques can incur high computational overhead during updates, HARDFS employs a small "cache" of *concrete states* for objects being actively modified (*e.g.*, the metadata for a currently open file). State management is further described in Section 4.2.3.

- **Action verifier:** The action-verifier module detects faulty actions of the main version with a set of micro-checks. Using these checks, the sleeved system verifies every action of the hardened functionality before it impacts other components. We describe in detail the challenges of verifying actions in Section 4.2.4.

- **Recovery module:** After a fault has been identified by a sleeved system, the recovery module is triggered. Since faulty behavior has been isolated within a single node, recovery can be as simple as crashing and rebooting the faulty node. However, rebooting can take a significant amount of time; therefore,

a sleeved system may optionally perform micro-recovery by semantically comparing every state object in the main version with the secondary version to recover only the corrupt objects. Recovery is described further in Section 4.2.5.

4.2.3 State Manager Module

We describe how the state manager operates, specifically how internal state is selected from the main version, derived from incoming messages and actions, and encoded in a lightweight manner.

Selective State Management

We selectively model a subset of the functionality and state of the main version. For instance, to verify namespace integrity (*e.g.*, correct file hierarchy) and corresponding operations (*e.g.*, file creation and deletion), HARDFS maintains directory entries without storing less important information such as access and modification times. State management is flexible: new information can be added incrementally to meet current needs (*e.g.*, one could add permission information for security checks if desired). HARDFS uses the same file formats for on-disk structures as vanilla HDFS, so upgrading HDFS to HARDFS or adding new memory state only requires a restart; copying data to a new file system is unnecessary.

In addition to storing the selected state, HARDFS needs logic for how state should be updated based on interposed messages; this logic acts as the second version. In order to implement this logic, we needed to understand the semantics of various protocol messages. For instance, for namespace management, upon a successful file creation message, HARDFS adds the corresponding file name to the maintained state.

Properly handling thread events that are periodically triggered is also necessary to keep both versions synchronized; if the second version were not aware of the thread events, it could not verify actions triggered by the threads. For example, when a periodic thread in the master node wakes and detects dead workers, the master may perform a block-replication action. The second version must be aware of this transition in order to verify the resulting actions correctly.

Lightweight State with Bloom Filters

We now discuss how our sleeved systems can manage state in an efficient and lightweight manner. While there are many ways to do this, in HARDFS, we use counting Bloom filters [40]. A Bloom filter is a probabilistic data structure that allows testing whether a data element is a member of a set. It is space efficient: the overhead does not depend on the state objects stored.

Our intuition for the use of Bloom filters is that sleeved systems typically only need to answer boolean questions (*e.g.*, does file `F` exist?) rather than answering non-boolean questions (*e.g.*, what are all files under directory `D`?). Thus, a Bloom filter is a fitting solution for compressing file-system metadata. The challenges that arise are dealing with non-boolean verification, excessive CPU overhead, and false positives. We describe our design to address these challenges.

- **Dealing with non-boolean verification:** Although using a Bloom filter is space efficient, one challenge is to represent non-boolean information, in particular information that changes and must be updated. For example, consider the case where both the main and second versions agree that file `F` is `100` bytes long. If a client appends the file and the worker tells the master that `F` is now `200` bytes long, then the second version must update its state

regarding `F`. However, the second version cannot overwrite the old entry {`F,100`} previously stored in the Bloom filter with a new entry {`F,200`}. Instead, it must perform two operations: delete the old entry {`F,100`}, and then insert the new entry {`F,200`}. To delete the old entry the second version must know the value of the old entry, but Bloom filters cannot answer non-boolean questions (in this example, what is the current length of `F`?).

To deal with this, we use an *ask-then-check* technique. That is, the secondary version asks a non-boolean question of the main version to determine the previous value for an entry before the main version's event handler executes. Because the returned result cannot be trusted, the second version then checks the previous value with a boolean question to the Bloom filter. In the above example, the secondary version first asks the main version for the length of `F` (which is `100`) and then checks via the Bloom filter that `F` is indeed `100` bytes long. With this verified and correct information, the secondary version performs the deletion (we use a counting Bloom filter to support this operation) and hence the overwrite.

- **Dealing with excessive CPU overhead:** While Bloom filters are space efficient, in some cases they can lead to excessive CPU overheads. To remedy this problem, HARDFS keeps a small "cache" of states being actively modified in *concrete* form (in contrast to the *compressed* form in the Bloom filter). In addition, HARDFS can optionally keep all data in concrete form, trading space efficiency for less CPU overhead. Finally, policies for converting data between concrete and compressed forms based on run-time measurements could be enforced.

- **Dealing with false positives:** The last challenge is the presence of false positives from two sources: Bloom filters and corrupted state or bugs in the sleeved code itself. First, Bloom filters fundamentally can return false positives [39]. A Bloom filter can "lie" that it contains file `F`, when in fact it does not. Fortunately, the false positive rate is relatively small and configurable. For instance, the probability of a false positive in a Bloom filter with 10 hash functions and 32 bits per data element is approximately 2 per million [83]. Doubling the number of bits per data element to 64 leads to a false positive rate of 4 per billion; at this rate, a cluster processing 100 operations/second would experience about one false positive per month.

  Second, the state maintained by the sleeved version itself can be corrupt due to memory problems or bugs. Fortunately, in crash-tolerant systems, false positives are benign from a correctness perspective because they only result in unnecessary recovery. The only danger is the degenerate case where a Bloom filter always generates a false positive for a particular element, resulting in repeated recovery. A small cache of concrete states solves this problem; the recovery mechanism remembers troublesome elements and pins them in the cache.

4.2.4 Action Verifier Module

The action verifier module detects incorrect actions performed by the main version that relate to the properties of interest. We classify incorrect actions into four types: corrupt, missing, orphan, and out-of-order. We believe it is important to detect all four types of incorrect actions. In our study of the HDFS bug reports (Section 3.1.2), we find that all these types of incorrect actions occur.

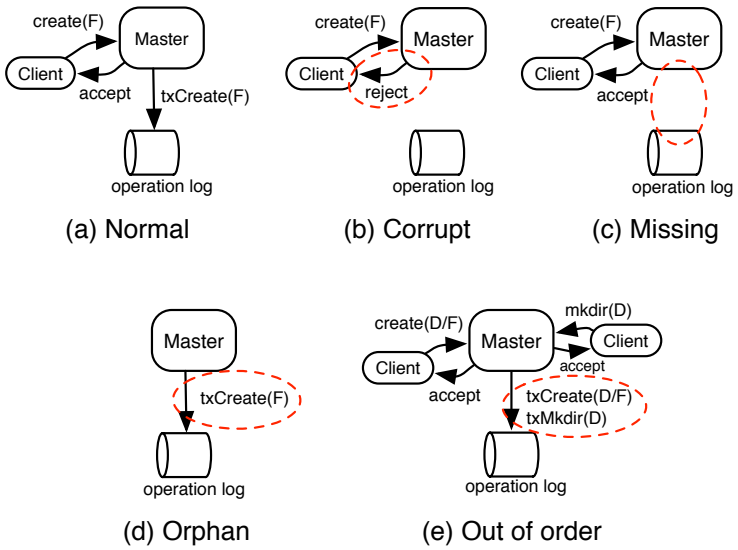

Figure 4.3: **Types of Incorrect Actions.** *(a) In an example of a correct action, the namenode accepts a file-creation request and writes the corresponding transaction to the on-disk operation log. However, the system could exhibit four types of incorrect actions: (b) corrupt actions such as the master incorrectly sends a wrong response, (c) missing actions such as a client request is accepted but no subsequent transaction is written, (d) orphan actions such as a transaction is written with no request origin, and (e) out-of-order actions such as transactions written in a wrong order.*

The following sections describe the four types of incorrect actions that could occur after a file creation request as illustrated in Figure 4.3. Figure 4.3a represents the correct behavior of a file creation; here the file does not exist, and thus the master accepts the request and writes an appropriate transaction to its persistent operation log.

Corrupt Actions

The first type of incorrect action is a corrupt action. Consider the scenario shown in Figure 4.3b where a client sends a request to create a file `F`; if the file did not

previously exist, then the request should be accepted. However, if the main version of the master behaves incorrectly (*e.g.*, the in-memory pathname is corrupted), then it will wrongly reject the request, while the second version accepts.

However, when there is *disagreement* between the secondary and main versions, the secondary version cannot be trusted to be the correct version. Thus, whenever disagreement occurs for any of the actions described below, the action verifier simply catches the incorrect action and takes additional steps to resolve the problem. These steps are described in more detail in Section 4.2.4.

Missing Actions

Missing actions represent the case where the main version should generate a specific action but fails to do so. For example, in Figure 4.3c, the master accepts the file creation request but forgets to write the corresponding transaction to the operation log.

To check for missing actions, the action verifier maintains an *expected action* list and generates expected actions for incoming requests or state changes that require a certain action. For example, a write to the operation log is expected to follow every accepted client-write. Expected-action entries describe both the action the main version should perform and when the action needs to be performed. Many actions are expected to occur before the main version's event handler returns, but in some cases, it is only possible to detect missing actions using timeouts. For example, since replication in HDFS is throttled, the namenode might not immediately send a replication command upon detecting an under-replicated block. Waiting too long, however, is an incorrect behavior that could lead to data loss.

Orphan Actions

Orphan actions represent the case where the main version performs unexpected actions. For instance, in Figure 4.3d, the master node writes to the operation log that file `F` is created although there is no origin for this request. To detect orphan actions, the action verifier leverages the expected action-list. Specifically, it signals an error when the action has no match in the expected-action list.

Orphan actions also cover the case of *duplicate* actions. For example, consider the block re-replication procedure due to dead worker nodes. If the master sends too many block re-replication commands, then the first re-replication command will be considered correct, while the subsequent ones will be considered orphans.

Out-of-order Actions

An action may depend on another one. For example, a transaction creating a new file `F` (*op2*) cannot precede the transaction making the parent directory `D` (*op1*). If the main version executes *op2* before *op1* (as in Figure 4.3e), the operation log will be corrupt, which may lead to severe consequences such as data loss or the master crashing permanently during checkpoint recovery. To address this challenge, action dependencies are tracked. However, tracking action dependencies is challenging and domain specific. We present an example of handling out-of-order actions in Section 4.3.1.

Handling Disagreement

By detecting incorrect actions as explained above, the action verifier can identify disagreements, but with only two versions to compare, it cannot known which ver-

sion is wrong; therefore, the action verifier resolves disagreements using domain-specific information and falls back on the safety of recovery from trusted state.

As an example, consider the request originally shown in Figure 4.3b, where the main and secondary versions disagree about the success of a file creation. It is entirely possible that the main version (correctly) rejected the request because a space quota was exceeded; if the second version does not incorporate knowledge about space quotas in its selective model, then it will (incorrectly) accept the request. Thus, the action verifier cannot conclude that the main version behaves incorrectly.

In several cases, we have found it much easier to implement a simplified secondary version that naively accepts requests that the complete main version rejects. To avoid false alarms in these cases, the action verifier examines the error code returned from the main version and ignores disagreements when the secondary version is not equipped to generate those cases. In the out-of-quota example, the action verifier agrees with the main version to reject the request and operation continues without recovery. Unfortunately, if the main version incorrectly reports “out-of-quota”, HARDFS will not detect it. There is a tradeoff: writing logic for more cases improves reliability, but increases engineering effort.

For some situations, the action verifier needs a mechanism to detect repeated disagreement. If a transient fault causes disagreement, the same discrepancy will not reappear after recovery, and normal operation will resume. However, one of the versions may have a bug that causes permanent disagreement. In this case, the developer is notified, and policy determines how to proceed until the code is fixed; entering HDFS safemode is one option (safemode prevents all file and block modifications).

4.2.5 Recovery Module

Once the action verifier detects a failure, the recovery module can apply many different techniques. One simple approach is crash and reboot (suitable for crash tolerant systems). A more fine-grained technique is micro-recovery where the recovery module pinpoints and recovers only the corrupt state. In addition to doing repair, the recovery module can also thwart destructive actions to prevent fault propagation to other nodes. We discuss the three techniques used by HARDFS below.

Reboot

Crash and reboot is a safe mechanism to prevent the propagation of corrupt state due to transient and non-deterministic failures. Upon reboot, the main version can safely reload its in-memory state from other trusted sources, such as persistent storage or other nodes across the network; the states of the secondary version are also reloaded since it interposes on these inputs.

We believe that it is appropriate to use rebooting for recovery when rebooting is quick and the faulty node is not a single point of failure. For instance, rebooting a worker node is appropriate since remaining workers can serve client requests and maintain data availability.

Micro-Recovery

When rebooting a node is expensive (*e.g.*, rebooting a master node may take hours [42]), a sleeved system can instead quickly identify and repair only the corrupted state. We call this technique micro-recovery, similar to micro-rebooting [54]. In micro-recovery, when a fault is detected, the node is frozen to prevent changes

to the system state. The recovery module then identifies the corrupted state by semantically comparing the secondary and main version state, and recovers it from a trusted source (*e.g.*, persistent storage).

For example, if the two versions disagree about the length of a file `F`, then micro-recovery reconstructs just `F`'s metadata from the checkpoint file and the operation log on disk. These sources can be trusted for two reasons: data is never written to them unless both versions agree, and solutions for preventing and detecting corruption to persistent storage are well known [31, 34, 38, 59, 90, 98, 106, 137, 141, 158, 159].

Disagreement can happen because of corruption in either secondary or main version state (or both). Repairing corrupt main version state is relatively easy because the recovery module can overwrite the corrupt state "in place". Repairing encoded state in a Bloom filter is more challenging. Consider a corrupted entry $\{\texttt{F,374}\}$, incorrectly indicating F is 374 bytes long. To repair the corrupted entry, the recovery module must delete the encoded entry and insert the correct entry, but it does not know that `F`'s length has been corrupted to 374. The solution described in Section 4.2.3 does not work because there is no entity that knows the corrupt value. Therefore, our solution is to begin with an empty Bloom filter instance and add entries as they are verified, either from main-version state or persistent storage, without a full reboot.

If micro-recovery does not find any disagreement, it means the detected faults might involve corruption in non-hardened functionality or bugs in the software logic, and thus the recovery module falls back to full reboot. If recovery is continuously repeated, an error report is generated as discussed in Section 4.2.4.

Thwarting Destructive Actions

Repairing a local node is of limited value if the faulty node causes permanent damage to other nodes before it recovers. HDFS workers send regular heartbeat messages to the master, and the master replies with messages directing workers to perform various actions. Some of these directives, such as “delete replica” or “decommission”, can cause irrecoverable data loss if misguided.

Our sleeved subsystems drop messages containing destructive directives if there is any disagreement between the main version and the secondary model about the objects in question. Our policy here is conservative; it is safer to potentially waste storage space than to risk deleting data unintentionally.

Overall, we find that micro-recovery is a powerful mechanism. However, unlike detection where the main version code does not need to be modified much, micro-recovery requires the main version be equipped with mechanisms for recovering specific states (*e.g.*, repairing a small subtree of the complete in-memory file hierarchy). We believe adding this machinery to existing systems is beneficial.

4.3 Implementation

In this section, we describe the specific details of the three HARDFS subsystems, HARDFS-N, HARDFS-R, and HARDFS-D, which harden the HDFS namespace management, replica management, and the read/write protocol of datanodes, respectively.

4.3.1 Namespace Management: HARDFS-N

Namespace management is a critical functionality in HDFS. The architecture of HDFS has a dedicated master, the *namenode,* which stores all file-system metadata in memory for fast operations. When the namenode executes a client request that changes the namespace, it writes an appropriate transaction to the on-disk operation log before responding back to the client. Periodically, the namenode replays this operation log to produce an on-disk checkpoint file that contains the complete namespace structure. HDFS splits files into 64MB blocks, which are replicated across *datanodes*.

To protect namespace integrity, HARDFS-N guards the in-memory namespace structures that are necessary for reaching data: the file-tree hierarchy, file-to-block mapping, and block-length information. With this protection, HARDFS-N detects namespace-related problems such as accidental file truncations, unreachable directories, and corrupt file-to-block mappings. When these problems are detected by HARDFS-N, faulty actions are not propagated to the client, persistent storage, or datanodes.

Maintaining State and Checking Actions

After interposing on incoming and outgoing messages, HARDFS-N can update its state (both Bloom filters and the expected-action list) and verify observed actions. Its logic for updating state from incoming messages is shown in Table 4.1. For example, in the first row of the table, at the entry path of the request `create(F)` from the client to the namenode, HARDFS-N records this fact by calling `insert(F)` to the Bloom filter. Table 4.1 can be seen as a concrete example of how a developer programs a sleeved service.

Message	Logic of the secondary version

`create(F)`
client requests NN to create file F
	`Entry:`
	`  If exists(F) Then reject;`
	`  Else`
	`  insert(F);`
	`  generateAction(txCreate[F]);`
	`Return: check response;`

`addBlk(F)`
client requests NN to allocate a block to file F
	`Entry: F:X = ask-then-check(F);`
	`Return:`
	`  B = addBlk(F);`
	`  If exists(F) & !exists(B) Then`
	`  X′ = X ∪ {B};`
	`  update(F:X, F:X′);`
	`  insert(B@0);`
	`  Else declare error;`

`blkRcvd(B,100)`
DN informs NN of received 100-byte block B
	`Entry:`
	`  B@L = ask-then-check(B);`
	`  update(B@L, B@100);`
	`Return: check response;`

`complete(F)`
client informs NN of write completion on file F
	`Entry:`
	`  If exists(F) Then`
	`  SIZES = empty-list`
	`  F:X = ask-then-check(F);`
	`  for B in X:`
	`  B@L = ask-then-check(B);`
	`  SIZES.append(B@L);`
	`  generateAction(txClose[F,SIZES]);`
	`Return: check response;`

Table 4.1: **SLEEVE for namespace management.** *The table shows how the secondary version derives the semantics of input messages from a client or datanode (DN) to the namenode (NN), manages its state using Bloom filter APIs, and generates expected actions.* `update(x1, x2)` *represents a* `delete(x1)` *followed by an* `insert(x2)`*; "Check response" means that the secondary version compares returned results and handles disagreement if any;* `F:B` *represents a mapping from file* `F` *to block* `B`*;* `B@x` *indicates that block* `B` *is* `x` *bytes long. Multiple Bloom filters (not shown) are used to encode different facts.*

HARDFS-N uses Bloom filters as a space-efficient data structure for encoding the file namespace. Only three APIs are needed: `insert(x)`, `delete(x)`, and `exists(x)`, where `x` is a variable-length byte array.

To encode a file hierarchy, the most straight-forward approach would be to perform `insert("d/f")` to indicate that there exists a directory `d` with a child `f`. However, this scheme leads to inefficient performance if a directory has many entries and is frequently renamed. Imagine there exist many entries `d/f1`, `d/f2`, `d/f3`, and so on, and directory `d` is renamed to `n`; since Bloom filters do not support overwrite (Section 4.2.3), the system would need to perform many ask-then-check operations to delete all `d/*` entries, and then insert all new `n/*` entries. Our solution is to introduce another level of indirection (*e.g.*, `keyOf(d)/f`). If the main version maintained a unique inode number for each directory, we could just use that information directly. Unfortunately, there is no such information. Instead, we use the hash code of the memory address for the Java object that represents the directory.

To catch orphan, missing, and out-of-order actions, HARDFS-N maintains an expected action list. For example, in the first row of Table 4.1, upon an incoming `create(F)` request, a future action `txCreate` is expected. To detect out-of-order transactions, HARDFS-N uses domain-specific knowledge. Specifically, a completed transaction (a successful `txCreate(D)`) implies that the associated object (`D`) is committed to the on-disk log and that subsequent child additions to `D` are also allowed. With this knowledge, out-of-order transactions can be detected (*e.g.*, if `txCreate(D/F)` is sent to the disk but `txCreate(D)` is still in the expected action list).

Recovery

HARDFS-N could recover from detected errors by rebooting the namenode and reconstructing all state. For faster recovery, HARDFS-N attempts micro-recovery first. Here, we describe further how corrupt states can be recovered from persistent storage.

HARDFS-N repairs corrupted states in memory (*e.g.*, bad `F`'s metadata) using states stored in the namenode's checkpoint file. Since we assume persistent storage is trusted (Section 4.2.5), the checkpoint file is expected to have "good" states. To obtain the latest checkpoint file, HARDFS-N forces the namenode to start a checkpointing process by replaying the operation log. However, HDFS checkpointing is relatively slow and I/O-intensive: it requires reading the old checkpoint file in its entirety, as well as the operation log, before it can write out a new checkpoint file. To optimize this, HARDFS-N avoids forcing a checkpoint when possible. HARDFS-N first scans the (relatively small) operation log to find the correct values for any of the relevant corrupted state (*e.g.*, `F`'s latest metadata). If no relevant transactions are found, HARDFS-N performs an efficient binary search on the checkpoint file for the needed information; the checkpoint file is already sorted based on pathname.

4.3.2 Replica Management: HARDFS-R

HDFS replica management involves the block-to-node mapping structure for tracking the node locations and number of replicas for every block. Since datanodes in a cluster may arrive and leave at any time, a block can be over- or under-replicated. Replica management ensures that each block has the intended number of replicas

by sending deletion and regeneration commands to different datanodes. When a block is created/regenerated, the datanode sends a `blkRcvd` message to the namenode. Every datanode also sends periodic `blockReport` messages containing the list of blocks managed by that datanode.

HARDFS-R hardens the namenode replica management functionality by protecting the integrity of block-mapping states (*e.g.*, no blocks will be accidentally deleted and no incorrect block locations will be returned to the client). Since many of the basics are similar to HARDFS-N, we focus on the differences.

Maintaining State and Checking Actions

HARDFS-R uses two Bloom filters to encode block-to-node mappings and replica-count information with simple formats such as `insert(BlkID:NodeID)` and `insert(BlkID:Count)`. For every block regeneration/deletion command sent by the main version, HARDFS-R performs various checks. For example, deleting a block replica should not make the block under-replicated; a regeneration command should only be performed on a valid block.

HDFS uses a periodic thread to detect dead nodes. When the thread is triggered, HARDFS-R is informed (Section 4.2.3) so that it can replicate the functionality for block accounting and manage its expected action list (*e.g.*, HARDFS-R expects to observe a block regeneration command if the block is under-replicated).

Recovery

In HARDFS-N, we saw that micro-recovery can be performed by reconstructing corrupt states from persistent storage. However, HDFS namenode does not maintain block-to-node maps in its persistent storage; therefore, full recovery is done

by requesting block mappings from all the datanodes (specifically by requesting `blockReport` commands). However, to perform micro-recovery of a corrupt block-to-node mapping (either in the main or secondary version), HARDFS-R only requests a block report from the corresponding node. If the datanode responds with matching information, then the mapping is trusted. Otherwise, HARDFS-R must fall back to full recovery by requesting block reports from all datanodes because there is no other way to determine the locations of the block.

4.3.3 Read/Write: HARDFS-D

Our final subsystem, HARDFS-D, hardens the datanode's metadata for reading and writing blocks. HARDFS-D can detect data access problems such as returning incorrect data or appending data at a wrong offset.

Managing State and Checking Actions

In each datanode, HARDFS-D protects two pieces of information: the list of blocks maintained by the datanode and the length of each block. In an append-only storage system such as HDFS, the block length is especially important since it defines the location of the next write; a corrupt length could lead to accidental overwrites. HARDFS-D uses two Bloom filters to protect the information (*e.g.*, `insert(B)` and `insert(B,100)`).

HARDFS-D verifies both disk and network actions. First, HARDFS-D checks that all disk accesses performed by the datanode are to the correct files and to the correct offsets. Second, HARDFS-D checks all outgoing network messages to ensure that any local corruption does not propagate to another datanode; this network check is vital because writes are preformed in a pipelined fashion.

Recovery

A corrupted and faulty datanode can be recovered with a simple reboot. Fortunately, because each block is typically replicated across multiple datanodes, rebooting a datanode does not affect data availability. In addition, as we will show in our evaluation, rebooting a datanode is fast, taking only a few seconds (Section 4.4.2). Therefore, we do not investigate micro-recovery for HARDFS-D.

4.4 Evaluation

We now evaluate HARDFS. Specifically, we present experimental results that answer the following questions:

- Is HARDFS effective at detecting and recovering from fail-silent faults caused by memory corruption and real-world bugs (Section 4.4.1)?
- How much time and space overhead does the additional bookkeeping incur? Does micro-recovery substantially improve recovery time (Section 4.4.2)?
- Does hardening HDFS require a reasonable amount of engineering effort (Section 4.4.3)?

4.4.1 Detection and Recovery

We evaluate the ability of HARDFS to detect faults and recover using three sets of experiments. We first randomly corrupt memory by injecting bit flips in the namenode's address space. To further understand the effect of such corruptions, we perform memory corruptions that target various fields in important data structures. Finally, we reintroduce real bugs to the codebase and measure how well HARDFS can prevent data loss.

Outcome	HDFS	HARDFS
No problem observed	728	460
Detect and reboot	-	140
Detect and micro-recover	-	107
Hang	22	16
Crash	133	268
Silent failure	**117**	**9**
(Corrupt pathname)	95	0
(Corrupt replication)	1	0
(Corrupt blocksize)	12	1
(Corrupt permission)	3	0
(Corrupt modification time)	6	8
Total	1000	1000

Table 4.2: **Outcomes of random memory corruption.** *This table presents the results of random memory corruption experiments in HDFS and HARDFS. The outcome of each experiment can be: no problem observed, corruption detected and system reboots, corruption detected and system performs micro recovery, hang, crash, and silent failure.*

Random Memory Corruption

We study how random memory corruptions affect the operation of vanilla HDFS and HARDFS by injecting random bit flips in the namenode's address space. Specifically, for each system we performed 1000 runs, each of which involved: (1) creating 10,000 files, (2) injecting random bit flips into the namenode's writable address space, and (3) recording if the system crashes or if `stat()` returns unexpected metadata (*i.e.*, a silent failure has occurred).

We focus on namenode corruptions because it is a single point of failure in the system and has more potential to propagate errors. Each bit has one chance in 50 million of being flipped. With our injection methodology, both the main implementation and the secondary model are subject to the random injections, so discrepancies can arise when state in either is corrupted.

Table 4.2 summarizes the experimental results. With HARDFS, the number of silent failures is reduced by a factor of 10 (from 117 to 9) because the failures are mostly detected and recovered (140 reboots and 107 micro-recovery instances). However, the JVM crashes twice as often (because additional bookkeeping increases the chance of pointer corruptions that lead to crashes). All the crashes were due to dangling pointers, memory protection errors, or illegal instructions. HARDFS trades availability for correctness and data safety.

The breakdown of silent failures illustrates the result of selective protection. In HDFS, corrupt pathnames are most common (95 cases) followed by corrupt blocksizes (12 cases). In contrast, the most common silent failure for HARDFS is a corruption of the modification time (8 cases), which arises because we selectively chose not to protect this inode field. For HARDFS, there is one dangerous failure, a corruption of blocksize; although HARDFS protects this field, it is possible that either our aggressive injections caused the logic checks to misbehave, or perhaps the same corruption happened to both the main version and secondary model, leading to this false negative. It is difficult to reproduce this case.

Targeted Memory Corruption

We also conduct targeted memory corruption because the random memory corruption experiment gives us little information about which part of memory is corrupted and its corresponding effect. To do this, we pick a field of the namespace data structure (*e.g.*, pathname, block ID, etc.), change it to an unexpected value (*e.g.*, from `f0` to `f1`), and run a simple workload (*e.g.*, file creation).

Table 4.3 summarizes our experimental results. We see that, despite employing on-disk replication, vanilla HDFS is quite fragile to memory corruption (many $\times$ and $\bigcirc$ outcomes). For instance, a block ID corruption can cause the namenode

	HDFS								HARDFS							
Message	P	C	S	R	B	I	G	L	P	C	S	R	B	I	G	L
mkdir	⊗	⊗	.	.	.	.	.	.	√	√	.	.	.	.	.	.
create	⊗[a]	⊗[b]	×	×	.	.	.	.	√	√	√	√	.	.	.	.
append	⊗[c]	⊗[c]	⊗[c]	⊗	⊗[c]	⊗[c]	⊗[b]	⊗[b]	√	√	√	√	√	√	√	√
addBlk	◯	◯	.	◯	.	.	.	.	√	√	.	√	.	.	.	.
abandonBlk	◯	◯	.	.	.	◯	◯	.	√	√	.	.	.	√	√	.
blkRcvd	.	.	.	.	.	⊗[b]	⊗[b]	.	.	.	.	.	.	√	√	.
fsync	⊗[a]	.	⊗[c]	⊗	⊗[c]	⊗[c]	⊗[b]	⊗[c]	√	.	√	√	√	√	√	√
complete	⊗[a]	.	⊗[c]	⊗	⊗[c]	⊗[c]	⊗[b]	⊗[c]	√	.	√	√	√	√	√	√
delete	⊗	⊗	.	.	⊗[b]	⊗[b]	.	.	√	√	.	.	√	√	.	.
rename	⊗	⊗	.	.	.	.	.	.	√	√	.	.	.	.	.	.
setRep	⊗[a]	.	.	×	.	.	.	.	√	.	.	√	.	.	.	.
setTimes	⊗[a]	.	.	.	.	.	.	.	√	.	.	.	.	.	.	.
getInfo	◯	◯	◯	◯	.	.	.	.	√	√	√	√	.	.	.	.
getListing	◯	◯	.	.	.	.	.	.	√	√	.	.	.	.	.	.
getBlks	⊗[a]	⊗	.	.	◯	◯	◯	◯	√	√	.	.	√	√	√	√

Table 4.3: **Namespace memory corruption experiments.** *The table shows results of corrupting namespace structure's fields for HDFS and HARDFS. Corrupted metadata fields are: (*P*) pathname, (*C*) child pointer, (*S*) default block size, (*R*) replication factor, (*B*) block pointer, (*I*) block ID, (*G*) block generation stamp, and (*L*) actual block length. Each cell presents the resulting actions from the combination of input message (e.g., mkdir) and corrupted internal state. Possible outcomes are: (×) faulty transaction, (◯) incorrect response, (√) correct transaction and response, (.) inapplicable. Footnotes:* [a] *the namenode fails to reboot and crashes permanently because of corrupted transaction log;* [b] *data loss;* [c] *inconsistency.*

	HDFS					HARDFS				
Message	I	G	O	D	C	I	G	O	D	C
setup_pipe	$\otimes$	$\otimes$	.	.	.	$\surd$	$\surd$	.	.	.
packet	.	.	$\oplus$	$\surd$	$\surd$	.	.	$\surd$	$\surd$	$\surd$
read	$\diamond$	$\surd$	$\diamond$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Table 4.4: **Corruption experiment with read/write protocol.** *The table shows results of corrupting internal replica metadata structure for HDFS and HARDFS. Corrupted metadata fields are: (*`I`*) block ID, (*`G`*) block generation stamp, (*`O`*) offset, (*`D`*) data, and (*`C`*) checksum. Each cell represents possible results observed by clients, including: ($\times$) data loss, ($\bigcirc$) temporary bad replica, ($+$) data corruption, ($\diamond$) bad data returned to clients, ($\surd$) success, and (.) inapplicable. The operations involve both a sending and receiving datanode, so for each case we run two experiments, injecting at each endpoint. Cells indicate the most severe result.*

to remove all replicas of a block; faulty transactions can be written to the on-disk operation log, eventually leading to unsuccessful checkpoints and reboots; corrupted states can propagate, leading to global corruption. HARDFS-N, on the other hand, correctly detects and recovers from all of the 54 injected faults without propagating the faults to the disk or other nodes.

We also investigate whether HARDFS-N can handle the secondary version behaving incorrectly. Specifically, we repeat the experiment in Table 4.3 where we corrupt the pathname and the file-to-block mapping, but this time within the Bloom filters. We find (not shown) that the recovery module successfully recreates HARDFS-N's internal state by reconstructing a new instance of the Bloom filter (as described in Section 4.2.5). The time to populate new instance of the Bloom filter is negligible: it takes only 2 seconds for a namespace of 200K files.

To measure the benefits of HARDFS-D, we corrupt replica metadata during block reads and writes (Table 4.4). Although vanilla HDFS datanodes handle faults better than the namenode in our last experiment, half of the trials still resulted in data loss, corruption, or incorrect responses. The data replication in

Bug	Year	Priority	Description
HADOOP-1135	2007	Major	Blocks in block report wrongly marked for deletion
HADOOP-3002	2008	Blocker	Blocks removed during safemode
HDFS-900	2010	Blocker	Valid replica deleted rather than corrupt replica
HDFS-1250	2010	Major	Namenode processes block report from dead datanode
HDFS-3087	2012	Critical	Decommission before replication during namenode restart

Table 4.5: **Software bugs.** *This table presents examples of HDFS bugs that cause incorrect behaviors. HARDFS is able to handle all of them and prevent data loss from happening.*

HDFS is useless if corruption can spread. HARDFS-D, however, detects faulty behaviors immediately and reboots the faulty node. In every trial, the fault is isolated, operations continue successfully, and no data is lost or corrupted.

Real Software Bugs

In this section, we explore how well HARDFS handles real software bugs. We chose five bugs from Hadoop and HDFS bug repositories [7, 10]; Table 4.5 gives a summary. The bugs have the following characteristics: they affect at least one of the subsystems we hardened, the bugs received a rank “major” or greater, and the bugs result in data loss under certain circumstances.

The bugs were discovered over a number of years, ranging from 2007 to 2012. The older bugs tend to be simple programmer oversights. For example, deletion of valid blocks can be triggered by a poorly written loop that processes block reports incorrectly [43], or because of missing safemode checks [47].

The newer bugs tend to be more subtle. HDFS-900 [45] is a rare race condition in the messaging protocol: if the namenode is in the middle of creating a new replica to replace a known corrupt replica, and a block report arrives at the

right time, the namenode will believe the block has an extra replica and trigger an extra delete, usually of a good replica. HDFS-1250 [48] represents another case where a block report arriving at the wrong time is problematic. If a datanode has died (detected when heartbeat messages have not arrived for some time), but the last block report from the datanode still somehow arrives at the namenode after some time, the namenode will mistakenly use the block report to compute replica counts, leading to under-replication. In HDFS-3087 [46], a datanode is being decommissioned, and the namenode is moving replicas to the remaining nodes. If the namenode is restarted in the middle of this process (perhaps due to a crash), the namenode upon restart will believe the datanode has no valid replicas because it has not yet received a block report, and will quickly finish the decommission process without moving all the data to the remaining nodes first.

For each bug, we made controlled modifications to our codebase to reproduce it. We limited ourselves to reintroducing the buggy code, injecting delays to reorder events, and dropping messages. As these bugs lead to behaviors that deviate from the expected model, HARDFS was able to detect the problem in each case and take appropriate action. In one case, HARDFS was able to restore proper state by restarting the namenode, and in four cases, HARDFS prevented data loss by simply thwarting the destructive directives (Section 4.2.5).

4.4.2 Efficiency

We now evaluate the performance impact, space overhead and recovery time of each HARDFS system. All experiments were conducted in a cluster of 21 machines, each having 8GB of memory and a 2.66GHz CPU.

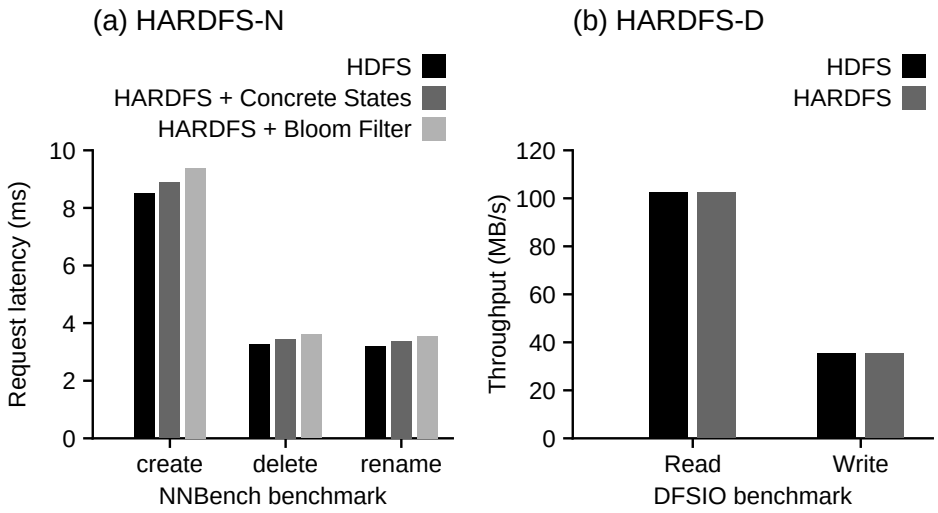

Figure 4.4: **Performance Overhead Evaluation.** *Figure (a) and Figure (b) show performance overhead of HARDFS-N and HARDFS-D, respectively. The performance overhead of HARDFS-R is negligible (not shown)*

Performance Impact

We evaluate the time overhead of HARDFS-N using the namenode benchmark (NNBench) in the Hadoop distribution (Figure 4.4a). This benchmark stresses many metadata requests by creating, renaming, and deleting files. HARDFS-N imposes acceptable overhead: 4% or 8%, with concrete state or Bloom filters, respectively. This performance impact appears acceptable.

In an experiment designed to stress HARDFS-R containing heavy client write activity and significant background processing of block reports, we found that the performance overhead of HARDFS-R is negligible. Finally, to evaluate the performance of HARDFS-D, we run the DFSIO benchmark with 3-way replication on a cluster containing one dedicated namenode and 20 datanodes and measure the average throughput of read and write operations. The results in Figure 4.4b show that the overhead of HARDFS-D is negligible for both workloads.

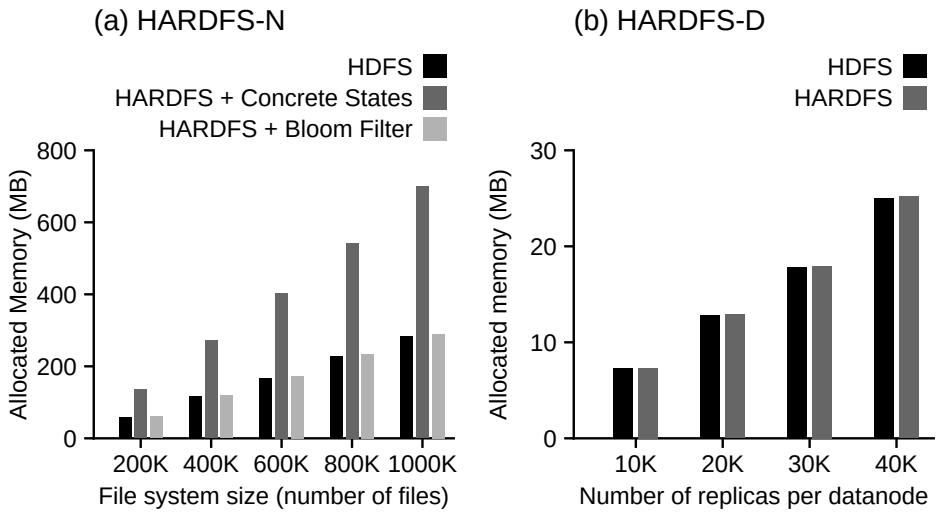

Figure 4.5: **Space Overhead Evaluation.** *These figures illustrate space overhead (i.e., the amount of memory allocated) for HARDFS-N and HARDFS-D based on the number of files in the system and the number of replicas per datanode. HARDFS-R with Bloom filters incurs less than 2% memory overhead (not shown).*

Memory Overhead

We measure the memory allocated for the namenode in both HDFS and HARDFS-N as the number of managed files is varied. Figure 4.5a shows that concrete states in HARDFS-N lead to memory overheads near 100%; this accentuates the need for lightweight data structures such as Bloom filters. As desired, the memory overhead of HARDFS with Bloom filters is negligible (2.6%).

We measure the memory allocated for both HDFS and HARDFS-D by varying the number of replicas a datanode manages (Figure 4.5b). The space efficiency of Bloom filters makes the memory overhead of HARDFS-D less than 1%. Finally, HARDFS-R with Bloom filters incurs less than 2% memory overhead.

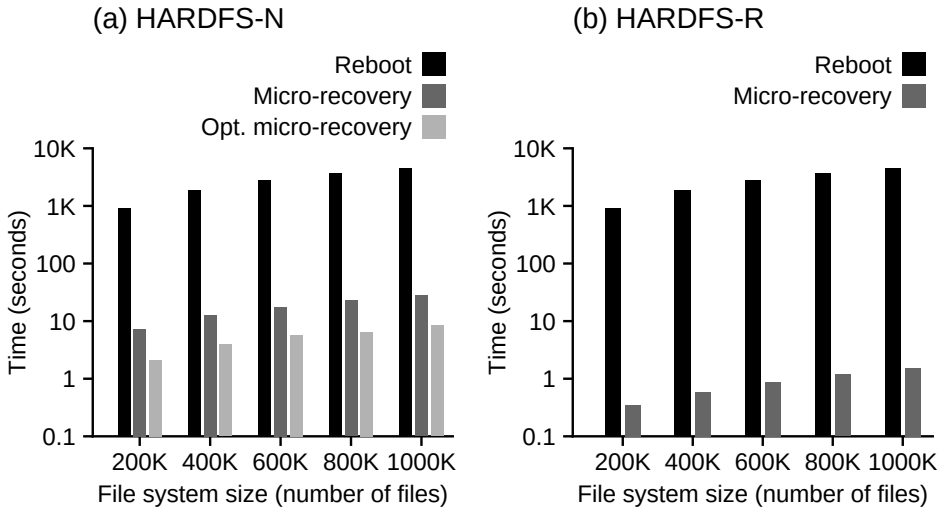

Figure 4.6: **Recovery Time Evaluation.** *These figures present the results of evaluating recovery time of HARDFS-N and HARDFS-R as the file system size increases. Overhead of recovery in HARDFS-D is negligible (not shown).*

Recovery Time

We measure the time for HARDFS-N to recover corrupt states using three approaches: simple reboot, micro-recovery, and optimized micro-recovery. Normal micro-recovery creates a new checkpoint based on the last checkpoint and the operation log; optimized micro-recovery computes only the needed state by efficiently scanning the log and last checkpoint (Section 4.3.1), thus avoiding the need to perform a complete checkpoint. All experiments were run in a cluster containing one namenode and 20 datanodes. Figure 4.6a summarizes the results. Although crash-and-reboot works correctly, it is prohibitively expensive: more than an hour is required to reboot a namenode managing 1 million files; most of this time is spent processing block reports from datanodes [42]. Fortunately, micro-recovery is highly efficient and more than two orders of magnitude faster. Our optimized version can recover corrupted state in less than 10 seconds, even

Subsystem	HDFS	HARDFS	
Namespace management	10114	1751	(17%)
Replica management	2342	934	(40%)
Read/Write protocol	5050	944	(19%)
Others	13339	0	(0%)
Total	30845	3629	(12%)

Table 4.6: **Engineering effort in lines of code.** *This table compares HDFS and HARDFS in terms of engineering effort. With the* SLEEVE *approach, HARDFS is much smaller than the original HDFS.*

when the namenode is managing 1 million files. Figure 4.6b shows similar benefits of using micro-recovery for HARDFS-R. Again, although a full reboot always behaves correctly, it can be prohibitively time consuming, requiring many hours. Micro-recovery can often reconstruct corrupted block map state from data nodes in less than one second. Clearly, micro-recovery should be used to repair corrupted entries when possible.

HARDFS-D does not utilize micro-recovery, as a datanode reboot is relatively quick. A datanode reboot involves reading a block list from local disks and sending the list to the namenode. In our experiments, it takes about 2 seconds to reboot a datanode storing 40,000 blocks (around 2.5TB) of data (not shown).

4.4.3 Engineering Effort

Table 4.6 compares the effort of implementing HDFS to the effort of hardening HDFS. By selecting three key modules where correctness is most important, we were able to focus our efforts on 57% of the codebase. Our lightweight second versions are much smaller than the original versions (17% to 40% of the main-version sizes); overall, our changes only increase the codebase by 12%. Although implemented in Java, HARDFS could be implemented in declarative languages [19, 146] in order to further reduce the engineering effort.

4.5 Conclusion

Equipped with abundant crash-recovery mechanisms, distributed systems are built to crash; nevertheless, they are susceptible to fail-silent behaviors caused by memory corruption and software bugs. If not handled properly, this failure mode can lead to many undesirable consequences such as data loss and corruption.

We address this challenge by introducing SLEEVE, a new approach that encourages developers to harden their systems against fail-silent behaviors with minimal engineering effort. Central to our approach is the idea of building a lightweight version that protects important components of the systems. This lightweight version serves as a detection layer that detects faulty behaviors and turns them into crashes, thus leveraging the crash-recovery machinery of existing systems to recover. In certain situation, the extra bookkeeping information of the lightweight version can be leveraged to enable a more fine-grained recovery.

We illustrate the feasibility and usefulness of the SLEEVE approach by applying it to harden HDFS. Our results show that it can detect and recover from a wide range of fail-silent behaviors caused by memory corruption and software bugs. Although our results are specific to HDFS, the SLEEVE approach and its principles discussed in this chapter can readily be applied to other cloud systems.

Chapter 5

Limplock: Impact of Limpware on Scale-out Cloud Systems

In this chapter, we highlight one often-overlooked cause of performance failures: *limpware* – hardware whose performance degrades significantly compared to its specification. In Chapter 2, we present reports showing how disk and network performance can drop by orders of magnitude. From these reports, we also find that unmanaged limpware can lead to *limplock*, a situation where a system progresses extremely slowly due to limpware and is not capable of failing over to healthy components (*i.e.*, the system enters and cannot exit from limping mode).

We target an important question in this chapter: what is the impact of limpware on today's scale-out systems? To answer this question, we assemble *limpbench*, a set of benchmarks that combine data-intensive load and limpware injections (*e.g.*, a degraded NIC or disk), to analyze five popular and varied large-scale cloud systems (Hadoop, HDFS, ZooKeeper, Cassandra, and HBase). With this analysis, we unearth distributed protocols and system designs that are susceptible to limplock. We also show how limplock can cascade in these systems.

This chapter is structured as follows. We first present the concept of limplock and its three subclasses: operation, node, and cluster limplock, along with system designs that allow them to happen in Section 5.1. We then describe limpbench for Hadoop, HDFS, ZooKeeper, Cassandra, and HBase in Section 5.2. In total, we have run 56 experiments that benchmark 22 protocols with limpware, for a total of almost 8 hours under normal scenarios and 207 hours under limpware scenarios. We present in detail our findings in Section 5.3. For each system, we present the impacts of limpware on the system and the design deficiencies. Overall, we find 15 protocols that can exhibit limplock.

5.1 Limplock

To measure the system-level impacts of limpware, we assembled limpbench (Section 5.2). From our findings of running limpbench, we introduce a new of concept of *limplock*, a situation where a system progresses slowly due to limpware and is not capable of failing over to healthy components (*i.e.*, the system enters and cannot exit from limping mode). We observe that limpware does not just affect the operations running on it, but also other healthy nodes, or even worse, a whole cluster. To classify such cascading failures, we introduce three levels of limplock: *operation*, *node*, and *cluster*. Below, we describe each limplock level. In each level, we dissect system designs that allow limplock to occur and escalate, based on our analysis of our target systems. The complete results will be presented in Section 5.3.

5.1.1 Operation Limplock

The smallest measure of limplock is operation limplock. Let us consider a 3-node write pipeline where one of the nodes has a degraded NIC. In the absence of limpware detection and failover recovery, the data transfer will slow down and enter limplock. We uncover three system designs that allow operation limplock:

- **Coarse-grained timeout:** Timeout is a form of performance failure detection, but a coarse-grained timeout does not help. For example, in HDFS, we observe that a large chunk of data is transferred in 64-KB packets, and a timeout is only thrown in the absence of a packet response for 60 seconds. This implies that limpware can limp to almost 1 KB/s without triggering a failover (Section 5.3.2). This read/write limplock brings negative implications to high-level software such as Hadoop and HBase that run on HDFS (Section 5.3.1 and Section 5.3.5).

- **Single point of failure (SPOF):** Limpware can be failed over if there is another resource or data source (*i.e.*, the *No SPOF* principle). However, this principle is not always upheld. For example, because of performance reasons, Hadoop intermediate data is not replicated. Here, we find that a mapper with a degraded NIC can make all reducers of the job enter limplock, and surprisingly, the speculative execution does not work in this case (Section 5.3.1). Another example is SPOF due to indirection (*e.g.*, HBase on HDFS). To access a data region, a client must go through the one HBase server that manages the region (although the data is replicated in three nodes in HDFS). If this *gateway* server has limpware, then it becomes a performance SPOF (Section 5.3.5).

- **Memoryless/revokeless retry:** A timeout is typically followed by a retry, and ideally a retry should not involve the same limpware. Yet, we find cases of prolonged limplock due to *memoryless* retry, a retry that does not use any information from a previously failed operation. We also find cases of revokeless retry. Here, a retry does not revoke previous operations. Under resource exhaustion, the retry cannot proceed.

5.1.2 Node Limplock

Node limplock is a situation where operations that must be served by this node experience a limplock, *although* the operations do *not* involve limpware. As an illustration, let's consider a node with a degraded disk. In-memory read operations served by this node should not experience a limplock. But, if the node is in limplock, the reads will also be affected. We also emphasize that node limplock can happen on *other* nodes that do *not* contain limpware (*i.e.*, a cascading effect). For example, let us consider a node A communicating with a node B that has limpware. If all communicating threads in A are in limplock due to B, then A exhibits node limplock, during which A cannot serve requests from/to other nodes. Node limplock leads to resource underutilization as the node cannot be used for other purposes. We uncover two system designs that can cascade operation limplock to node limplock:

- **Bounded multi-purpose thread/queue:** Developers often use a bounded pool of multi-purpose threads where each thread can be used for different types of operation (*e.g.*, read and write). Here, if limplocked operations occupy all the threads, then the resource is exhausted, and the node enters limplock. For example, in HDFS, limplocked writes to a slow disk can

occupy all the request threads at the master node, and thus in-memory reads are affected (Section 5.3.2). Similarly, a multi-purpose queue is often used. Here, a node uses one queue to communicate to all other nodes, and hence a slow communication between a pair of nodes can make the single queue full, disabling communication with other nodes.

- **Unbounded thread/queue:** Unbounded solutions can also lead to node limplock due to backlogs. For example, in the ZooKeeper quorum protocol, a slow follower can make the leader's in-memory queue grow as the follower cannot catch up with all the updates. Over time, this backlog will exhaust the leader's memory space and lead to node limplock (Section 5.3.3).

5.1.3 Cluster Limplock

Cluster limplock is the most severe level of limplock. Here, a single piece of limpware makes the whole cluster performance collapse. This condition is different from node limplock where only one or a subset of all the nodes are affected. Distributed systems are prone to cluster limplock as nodes communicate with each other via which limplock cascades. There are two scenarios that lead to cluster limplock.

- **All nodes in limplock:** If all nodes in the system enter limplock, then the cluster is technically in limplock. This happens in protocols with a small maximum number of resources. For example, in Hadoop, by default a node can have two map and reduce tasks. We find that a wide fan-out (*e.g.*, many reducers reading from a slow mapper) can quickly cause cluster limplock (Section 5.3.1).

- **Master-slave architecture:** This architecture is prone to cluster limplock. If a master exhibits node limplock, then entire operations that are routed to the master will enter limplock. In cloud systems where the slave-to-master ratio is typically large (*e.g.*, HDFS), master limplock can make many slave nodes underutilized.

5.2 Limpbench

We now present limpbench, a set of benchmarks that we assembled for two purposes: to quantify limplock in current cloud systems and to unearth system designs leading to limplock. We have benchmarked the impact of limpware on five scale-out systems (Hadoop/HDFS-1.0.4, ZooKeeper-3.4.5, Cassandra-1.2.3, and HBase-0.94.2). Our results are shown in Table 5.1 where each row in represents an experiment. In each experiment, we target a particular protocol, run a microbenchmark, and inject a slow NIC/disk. We run our experiments on the Emulab testbed [177]. Each experiment is repeated 3-5 times. Figure 5.1 shows the empirical results and will be described in Section 5.3.

5.2.1 Methodology

Each experiment includes four important components: data-intensive protocols, load stress, fault injections (limpware and crash), and white-box metrics.

- **Data-intensive protocols.** We evaluate data-intensive system protocols such as read, write, rebalancing, but not background protocols like gossipers. In some experiments, we mix protocols that require different resources (*e.g.*, read from cache, write to disk) to analyze cascades of limplock. Our target protocols are listed in the “Protocol” column of Table 5.1.

ID	Protocol	Limp-ware	Injected Node	Workload	Base Latency	OL	NL	CL
H1	Speculative execution	Net	Mapper	WordCount: 512 MB dataset	80	√	.	.
H2	Speculative execution	Net	Reducer	WordCount: 512 MB dataset	80	.	.	.
H3	Speculative execution	Net	Job Tracker	WordCount: 512 MB dataset	80	.	.	.
H4	Speculative execution	Net	Task Node	1000-task Facebook workload	4320	√	√	√
F1	Logging	Disk	Master	Create 8000 empty files	12	√	√	√
F2	Write	Disk	Data	Create 30 64-MB files	182	.	.	.
F3	Read	Disk	Data	Read 30 64-MB files	120	.	.	.
F4	Metadata Read/Logging	Disk	Master	Stats 1000 files + heavy updates	9	√	√	√
F5	Checkpoint	Disk	Secondary	Checkpoint 60K transactions	39	√	.	.
F6	Write	Net	Data	Create 30 64-MB files	208	√	.	.
F7	Read	Net	Data	Read 30 64-MB files	104	√	.	.
F8	Regeneration	Net	Data	Regenerate 90 blocks	432	√	√	√
F9	Regeneration	Net	Data-S/Data-D	Scale replication factor from 2 to 4	11	√	.	.
F10	Balancing	Net	Data-O/Data-U	Move 3.47 GB of data	4105	√	.	.
F11	Decommission	Net	Data-L/Data-R	Decommission a node having 90 blocks	174	√	√	√
Z1	Get	Net	Leader	Get 7000 1-KB znodes	4	.	.	.
Z2	Get	Net	Follower	Get 7000 1-KB znodes	5	.	.	.
Z3	Set	Net	Leader	Set 7000 1-KB znodes	23	√	√	√
Z4	Set	Net	Follower	Set 7000 1-KB znodes	26	.	.	.
Z5	Set	Net	Follower	Set 20KB data 6000 times to 100 znodes	64	√	√	√
C1	Put (quorum)	Net	Data	Put 240K KeyValues	66	.	.	.
C2	Get (quorum)	Net	Data	Get 45K KeyValues	73	.	.	.
C3	Get (one) + Put (all)	Net	Data	Get 45K KeyValues + heavy puts	36	.	.	.
B1	Put	Net	Region Server	Put 300K KeyValues	61	√	.	.
B2	Get	Net	Region Server	Get 300K KeyValues	151	√	.	.
B3	Scan	Net	Region Server	Scan 300K KeyValues	17	√	.	.
B4	Cache Get/Put	Net	Data-H	Get 100 KeyValues + heavy puts	4	√	√	.
B5	Compaction	Net	Region Server	Compact 4 100-MB sstables	122	√	√	.

Table 5.1: **Limpbench Experiments.** *Each table entry represents an experiment in limpbench. H, F, Z, C, and B in the "ID" column represent **H**adoop, HD**F**S, **Z**ooKeeper, **C**assandra, and H**B**ase. The columns describe the experiment ID, the target protocol being tested, the limpware type (disk/network), the node type where the limpware is injected, the workload, and the base latency of the experiment under no limpware. A tick mark in OL, NL, and CL columns implies that the experiment leads to operation, node, and cluster limplock respectively. Data: datanode; Data-H: datanode storing HLog; Data-S, Data-D, Data-O, Data-U, Data-L and Data-R represent source, destination, over-utilized, under-utilized, leaving, and remaining datanodes, respectively.*

- **Load stress.** We construct microbenchmarks that stress request load (listed in the "Workload" column of Table 5.1). Performance failures often happen under system load. Each benchmark saturates 30-70% of the maximum throughput of the setup.

- **Fault Injection.** First, we perform *limpware injection* on a local network card (NIC) or disk. We focus on I/Os as they can slow down by orders of magnitude. The perfect network and disk throughputs are 100 Mbps and 80 MB/s respectively. In each experiment, we inject *three limpware scenarios* (slow down a NIC/disk by 10x, 100x, and 1000x). We only inject slow disk on experiments that involve synchronous writes to disk. Many protocols of our target systems only write data to buffer cache, and hence the majority of the experiments involve slow NIC. We also perform *node-aware* limpware injection; across different experiments, limpware is injected on different types of node (*e.g.*, master vs. datanode).

  Second, we perform *crash injection*, mainly for two purposes: to show the duration of crash failover recovery (*e.g.*, write failover, block regeneration) and to analyze limpware impacts during fail-stop recovery (*e.g.*, limpware impact on a data regeneration process). The first purpose is to compare the speed of fail-stop recovery with limpware recovery (if any).

- **White-box metrics.** We monitor system-specific information such as the number of working threads and lengths of various request queues. We do not treat the systems as black boxes because limpware impact might be "hidden" behind external metrics such as response times. White-box monitoring helps us unearth hidden impacts and build better benchmarks.

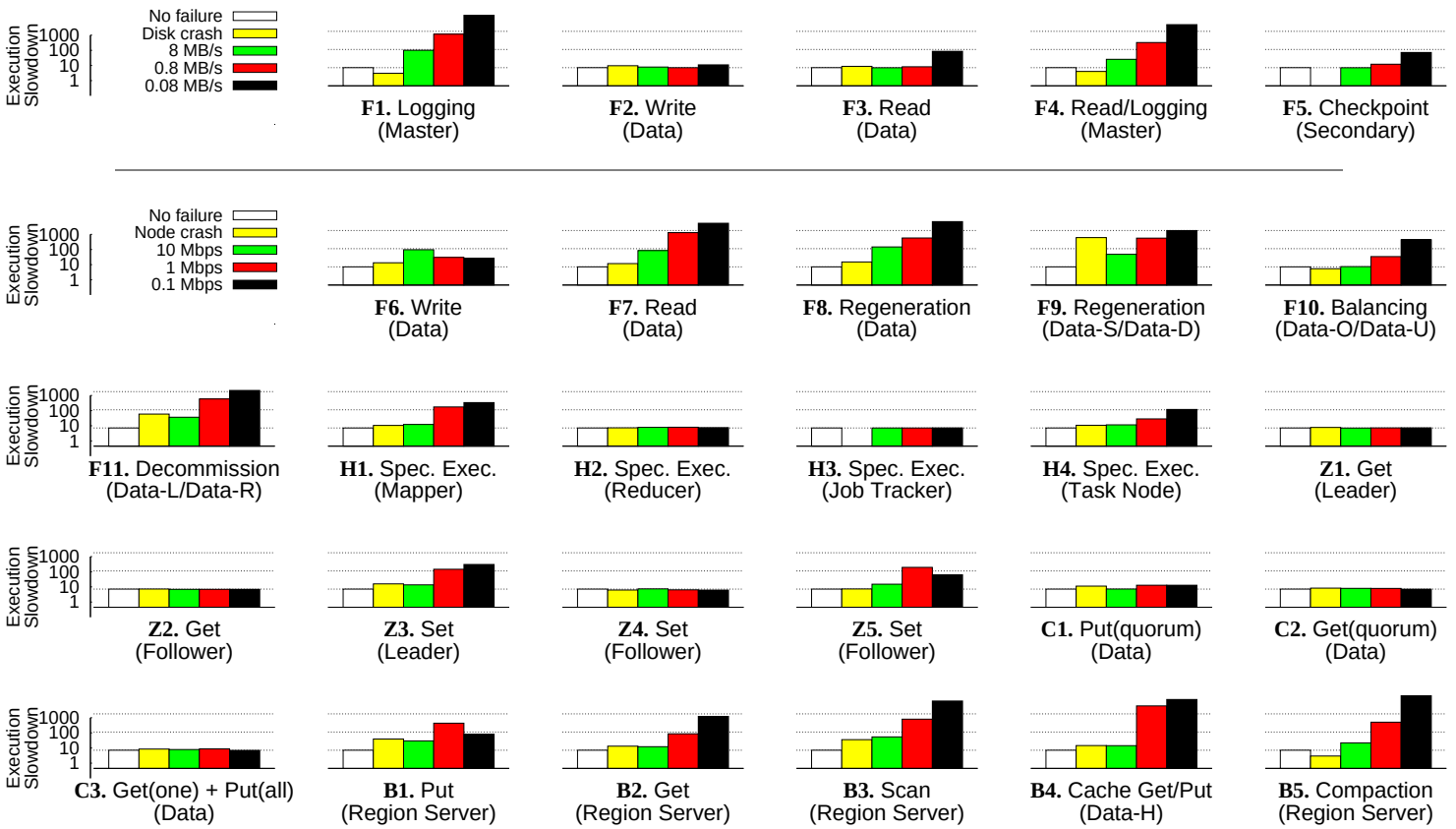

Figure 5.1: **Limpbench Results.** *Each graph represents the result of each experiment (e.g., F1) described in Table 5.1. The y-axis plots the slowdowns (in log scale) of an experiment under various limpware scenarios. In the first row, a slow disk is injected. In the rest, a slow NIC is injected. The graphs show that cloud systems are crash tolerant, but not limpware tolerant.*

5.3 Results

We now present the results of running limpbench on Hadoop, HDFS, ZooKeeper, Cassandra, and HBase. The columns OL, NL, and CL in Table 5.1 label which experiments/protocols exhibit operation, node, and cluster limplock respectively. We will not discuss individual experiments but rather focus our discussion on how and why these protocols exhibit limplock. Figure 5.1 quantifies the impact of limplock in each experiment. Uphill bars (*e.g.*, in F1, H1) imply the experiment observes a limplock. Flat bars (*e.g.*, in F2, H2) imply otherwise. Up-and-down bars (*e.g.*, in B1, Z5) imply that when congestion is severe (*e.g.*, 0.1 Mbps NIC), the connection to the affected node is "flapping" (connected and disconnected continuously), and the cluster performs better but not optimally. In the following sections, major findings are written in italic text. Section 5.3.6 summarizes our high-level findings and lessons learned.

5.3.1 Hadoop

In Hadoop, there is one job tracker which manages jobs running on slave nodes. Each job is divided into a set of map and reduce tasks. A mapper processes an input chunk from HDFS and outputs a list of key-value pairs. When all mappers have finished, each reducer fetches its portion of the map outputs, runs a reduce function, and writes the output to HDFS files. At the heart of Hadoop is speculative execution, a protocol that monitors task progress, detects stragglers, and runs backup tasks to minimize job execution time. We evaluate the robustness of the default Hadoop speculation (LATE [187]) by injecting a degraded NIC on three kinds of nodes: job tracker, map and reduce nodes (to simplify diagnosis, we do not collocate mappers and reducers). In our discussion below, the term "slow map/reduce node" implies a map/reduce node that has a degraded NIC.

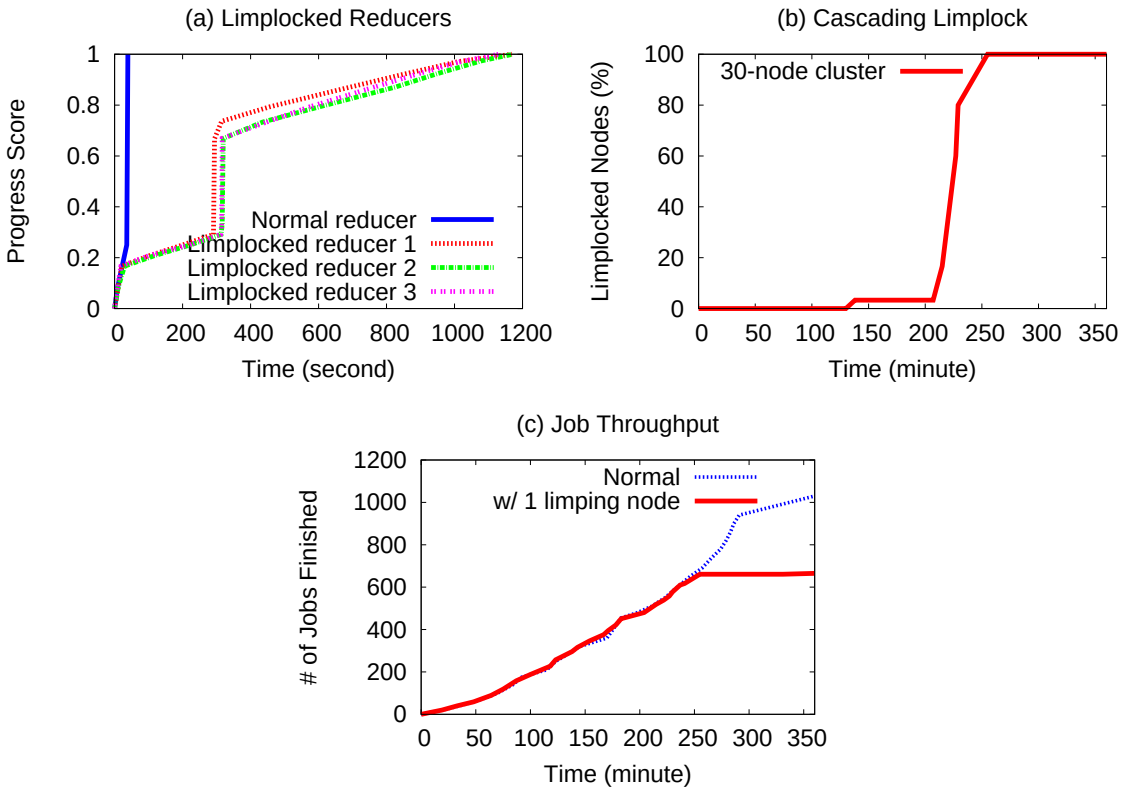

Figure 5.2: **Hadoop Limplock.** *The graphs show (a) the progress scores of limplocked reducers of a job in experiment H1 (a normal reducer is shown for comparison), (b) cascades of node limplock due to single limpware, and (c) a throughput collapse of a Hadoop cluster due to limpware. For Figures (b) and (c), we ran a Facebook workload [60] on a 30-node cluster.*

Limplock Free

We find that Hadoop speculation works as expected when a reduce node is slow (Figure 5.1, H2). Here, the affected reduce tasks are re-executed on other nodes. Slow job tracker also does not affect job execution time as it does not perform data-intensive tasks (H3).

Operation Limplock

We find three scenarios where Hadoop speculation is not immune to slow network: when (1) a map node is slow, (2) all reducers experience HDFS write limplock, and (3) both original and backup mappers read from a remote slow node. These cases lead to job execution slowdowns by orders of magnitude (*e.g.*, H1). The root causes of the problem are imprecise straggler detection and intra-job speculation. We now discuss these three limplock scenarios.

First, *a slow map node can slow down all reducers of the same job, and hence does not trigger speculation*. We find that a mapper can run on a slow node without being marked as a straggler. This is because the input/output of a mapper typically involves only the local disk due to data locality; the slow NIC does not affect the mapper. However, during the reduce phase, the implication is severe: when all reducers of the same job fetch the mapper's output through the slow link, *all* of the reducers progress at the same slow rate. Speculation is not triggered because a reduce task is marked as a straggler only if it makes little progress relative to others of the *same* job.

Although it is the reducers that are affected, re-executing the reducers is not the solution. We constructed a synthetic job where only a subset of the reducers fetch data from a slow map node. The affected reducers are re-executed, however,

the backup reducers still read from the same slow source again. The slow map node is a single point of performance failure, and the solution is to rerun the map task elsewhere.

Second, *all reducers of a job can exhibit HDFS write limplock, which does not trigger speculation*; HDFS write limplock is explained in Section 5.3.2. The essence is that *all* reducers write their outputs at the same slow rate, similar as the previous scenario. To illustrate these two scenarios, Figure 5.2a plots the progress scores of three reducers of a job. There are three significant regions: all scores initially progress slowly due to a slow input, then jump quickly in computation mode, and slow down again due to HDFS write limplock. These 40-second tasks finish after 1200 seconds due to limplock.

Finally, *both original and backup mappers can be in limplock when both read from a remote slow node via HDFS*. There are two underlying issues. First, to prevent thrashing, Hadoop limits the number of backup task (default is one); if a backup task exhibits limplock, so does the job. Second, although HDFS employs three-way replication, HDFS can pick the same slow node several times. This is a case of *memoryless retry*. That is, Hadoop does not inform HDFS that it wants a different source than the previous slow one.

Node Limplock

Node limplock occurs when its task slots are all occupied by limplocked tasks. A Hadoop node has a limited number of map and reduce slots. If all slots are occupied by limplocked tasks, then the node will be in limplock. The node is underutilized as it cannot run other healthy jobs unaffected by the limpware.

Cluster Limplock

A single piece of limpware can cripple an entire Hadoop cluster. This happens when limpware causes limplocked map/reduce tasks, which then lead to limplocked nodes and eventually a limplocked cluster when all nodes are in limplock. To illustrate this in real settings, we ran a Facebook workload [60] on a 30-node cluster with a node that has a degraded NIC. Figure 5.2b shows how limpware can cause many nodes to enter limplock over time and eventually cluster limplock. Figure 5.2c shows how the cluster is underutilized. In a normal scenario, the 30-node cluster finishes around 172 jobs/hour. However, under cluster limplock, the throughput collapses to almost 1 job/hour at t = 250 minutes.

5.3.2 HDFS

HDFS employs a dedicated master and multiple datanodes. The master serves metadata reads and writes with a fixed-size thread pool. All metadata is kept in memory for fast reads. A logging protocol writes metadata updates to an on-disk log that is replicated on three storage volumes. Datanodes serve data read and write requests. A data file is stored in 64-MB blocks. A new data block is written through a pipeline of three nodes by default. Dead datanodes will lead to under-replicated blocks, which then will trigger a block regeneration process. To reduce noise to foreground tasks, each data-node can only run two regeneration threads at a time with a throttled bandwidth. Block regeneration is also triggered when a datanode is decommissioned or users increase file replication factor on the fly. HDFS also provides a rebalancing process for balancing disk usage across datanodes.

We evaluate the robustness of HDFS protocols by injecting a degraded disk or NIC. To evaluate master protocols, we inject a degraded disk out of three available disks, but not NIC because there is only a single master.

Limplock Free

Datanode-related protocols are in general immune to slow disk. This is because data writes only flush updates to the OS buffer cache; on-disk flush happens in the background every 30 seconds. In our experiments, with 512 MB RAM, the write rate must be above 17 MB/s to reveal any impact. The network however is limited to only 12.5 MB/s.

Operation Limplock

HDFS is built for fail-stop tolerance but not limpware tolerance; we find numerous protocols that can exhibit limplock due to a degraded disk or NIC, as shown in Table 5.1. Below we frame our findings in terms of HDFS design deficiencies that lead to limplock-prone protocols: coarse-grained timeouts, memoryless and revokeless retries, and timeout-less protocols.

First, *HDFS data read and write protocols employ a coarse-grained timeout such that a NIC that limps above 1 KB/s will not trigger a failover* (Figure 5.1, F7). Specifically, these protocols transfer a large data block in 64-KB packets, and a timeout is only thrown in the absence of a packet response for 60 seconds. HDFS might expect that upper-level software employs a more fine-grained domain-specific timeout, however, such is not the case in Hadoop (Section 5.3.1) and HBase (Section 5.3.5). Timeouts based on relative performance [25] might be more appropriate than constant long timeouts.

Symbol	Definition/Derivation
n	# nodes
b	# blocks per node / to regenerate
r	# user requests
P_{rl}	$1 - (\frac{n-1}{n})^r$
P_{wl}	$1 - (\frac{n-3}{n})^r$
P_{nl}	$1 - (\frac{n-3}{n-2})^{\frac{b}{n-1}} - \frac{b}{(n-1)\times(n-2)} \times (\frac{n-3}{n-2})^{\frac{b-n+1}{n-1}}$
P_{cl}	${P_{nl}}^{n-2}$
$p_{nl}(i)$	$\binom{n-2}{i} \times {P_{nl}}^i \times (1-P_{nl})^{n-2-i}$
p_{bl}	$\sum_{i=2}^{n-2} p_{nl}(i) \times \frac{\binom{i}{2}}{\binom{n-1}{2}} + \sum_{i=1}^{n-2} p_{nl}(i) \times \frac{i}{\binom{n-1}{2}}$
P_{bl}	$1 - (1-p_{bl})^b$

Table 5.2: **HDFS Limplock Frequency.** *The table shows the probabilities of a user experiences at least one read (P_{rl}) and write limplock (P_{wl}), a node is in regeneration limplock (P_{nl}), a cluster is in regeneration limplock (P_{cl}), exactly i nodes are in regeneration limplock ($p_{nl}(i)$), a block is in regeneration limplock (p_{bl}), and at least one block is in regeneration limplock (P_{bl}). The detailed derivation is presented in Appendix A.*

Second, even in the presence of timeouts, *multiple retries can exhibit limplock due to memoryless retry*, similar to the Hadoop case. Here, limpware is involved again in recovery. Consider a file scale-up experiment from 2 to 4 replicas, and one of the source nodes is slow. Even with random selection, HDFS can choose the same slow source multiple times (F9).

Finally and surprisingly, *some protocols are timeoutless*. For example, the log protocol at the master writes to three storage volumes serially without any timeout. One degraded disk will slow down all log updates (F1). We believe this is because applications expect the OS to return some error code if hardware fails, but such is not the case for limpware.

To show how often operation limplock happens in HDFS, we model HDFS

basic protocols (read, write, and regeneration), derive their limplock probabilities as shown in Table 5.2, and use simulations to confirm our results. The details of this calculation are presented in Appendix A. Figure 5.3a and 5.3b plot the probabilities of a user to experience at least one read and write limplock respectively as a function of cluster size and request count. The probabilities are relatively high for a small to medium cluster (*e.g.*, 30-node). This significantly affects HDFS users (*e.g.*, Hadoop operation limplock described in Section 5.3.1).

Node Limplock

We find two protocols that can lead to node limplock: regeneration and logging. The root cause is resource exhaustion by limplocked operations.

HDFS can exhibit regeneration node limplock where all the node's regeneration threads are in limplock. Regeneration is run by the master for under-replicated blocks. For each block, the master chooses a source that has a surviving replica and a destination node. A source node can only run two regeneration threads at a time. Thus, regeneration node limplock occurs if the source node has a slow NIC or when the master picks a slow destination for both threads. Here, the node's regeneration resources are all exhausted.

Regeneration node limplock cannot be unwound due to revokeless recovery. Interestingly, the master employs a timeout to "recover" a slow/failed regeneration process, however, it is revokeless; the recovery does not revoke the limplocked regeneration threads on the affected datanodes (it only implicitly revokes if the source/destination crashes). Therefore, as the master attempts to retry, the resources are still exhausted, and the retry fails silently.

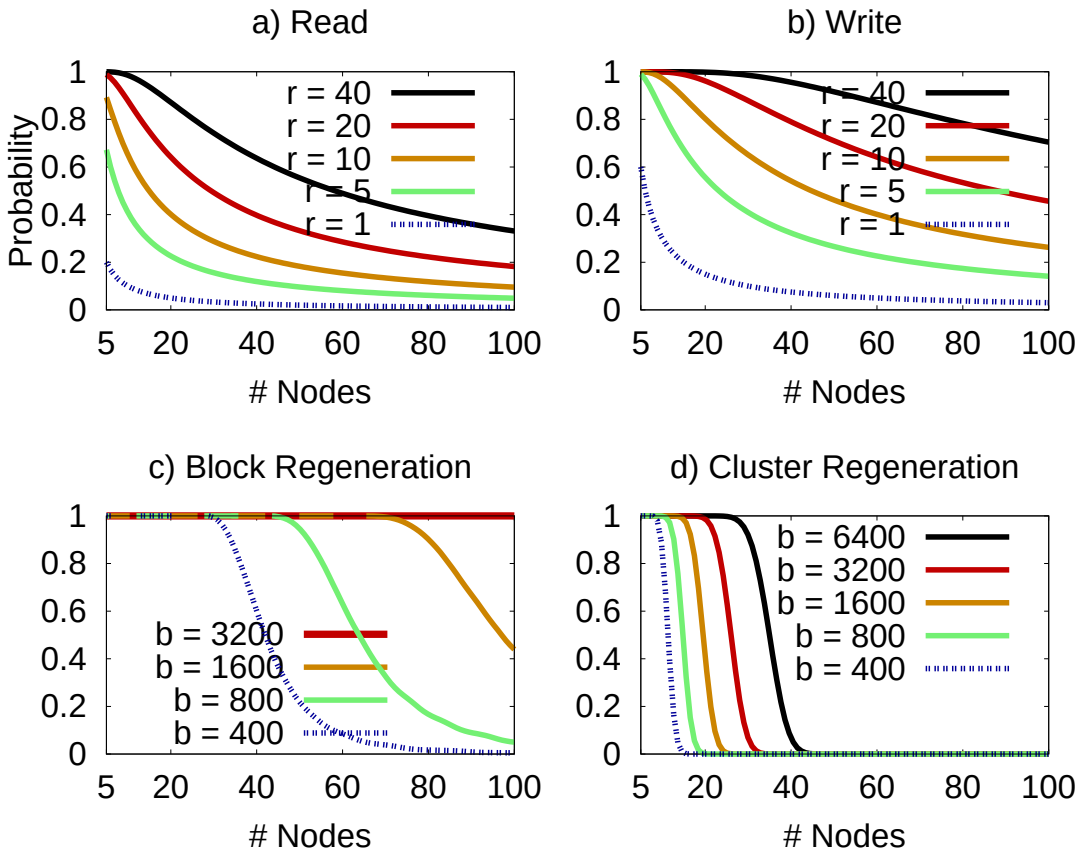

Figure 5.3: **HDFS Limplock Probabilities.** *The figures plot the probabilities of (a) read limplock/P_{rl}, (b) write limplock/P_{wl}, (c) block limplock/P_{bl}, (d) and cluster regeneration limplock/P_{cl}, as defined in Table 5.2. The x-axis plots cluster size.*

Regeneration node limplock prolongs MTTR and potentially decreases MTTDL. Nodes that exhibit regeneration limplock can be harmful because the nodes cannot be used as sources for regenerating other under-replicated blocks. This essentially prolongs the data recovery time (MTTR). In one experiment, a stable state (zero under-replicated block) is reached after 37 hours, 309x slower than in a normal case (Figure 5.1, F8). If more nodes die during this long recovery, some blocks can be completely lost, essentially shortening the mean time to data loss (MTTDL). To generalize this problem, we introduce a new term, *block limplock*, which is a scenario where at least an under-replicated block B cannot be regenerated (possibly for a long time) because the source nodes are in limplock. We derive the probability of at least one block limplock (P_{bl} in Table 5.2) as a function of cluster size and number of blocks stored in every datanode (which also represents the number of under-replicated blocks). Figure 5.3c plots this probability. The number is alarmingly high; even in a 100-node cluster, a dead 200-GB node (3200 under-replicated blocks) will lead to at least one block limplock.

Other than regeneration, we do not find any datanode protocols that cause node limplock, mainly because a datanode does not have a bounded thread pool for other operations (*e.g.*, it creates a new thread with small memory footprint for each data read/write). We however find a node limplock case in the master logging protocol. In master-slave architecture, master limplock essentially leads to cluster limplock, which we describe next.

Cluster Limplock

The two HDFS protocols that can exhibit node limplock, regeneration and logging, eventually lead to cluster limplock.

First, *an HDFS cluster can experience total regeneration limplock where all regeneration threads are in limplock.* As defined before, if all nodes are in limplock then the cluster is in limplock; all regeneration threads progress slowly and affect the MTTR and MTTDL. Figure 5.3d plots the probability of cluster regeneration limplock (P_{cl} in Table 5.2). A small cluster ($<$ 30 nodes) is prone to regeneration cluster limplock, as also confirmed in our simulation [78].

Second, *HDFS master is in limplock when all handlers are exhausted by limplocked log writes.* As discussed earlier, a slow disk at the master leads to limplocked log updates (F1). This leads to resource exhaustion under a high load of updates. This is because the master employs a fixed-size pool of multi-purpose threads for handling metadata read/write requests (default is 10). Therefore, as limplocked log writes occupy all the threads, incoming metadata read requests which only need to read in-memory metadata (do not involve the limpware) are blocked in a waiting queue. In one experiment, in-memory read throughput collapses by 233x (F4). Since the master is in node limplock, all operations that require metadata reads/writes essentially experience a cluster limplock.

5.3.3 ZooKeeper

ZooKeeper has a single leader and multiple follower nodes, and uses znodes as data abstraction. ZooKeeper basic APIs include create, set, get, delete, and sync. Znode get protocol is served by any node, but updates must be forwarded to the leader who executes a quorum-based atomic broadcast protocol to all the followers. If quorum is reached, ZooKeeper returns success. In our evaluation, we inject a degraded NIC on two types of nodes: leader and follower. The client always connects to a healthy follower.

Limplock Free

As the client connects to a healthy node, the get protocol is limplock free because the slow leader/follower is not involved (Figure 5.1, Z1, Z2). Similarly, based on one experiment (Z4), the quorum-based broadcast protocol is also limplock free. Here, a slow follower who has not yet committed updates does not affect the response time as the majority of the nodes are healthy.

Operation Limplock

If the leader is slow, all updates are in limplock (Z3). Leader-follower architecture must ensure that the leader is the most robust node. We notice that the slow leader can be congested and its IPC timeout disconnects all connections, which then triggers a leader election process. However, as data connections were cut, congestion diminishes, and the slow leader can join the election. In fact, this previous slow leader is likely to be elected again as the election favors a node that has the latest epoch time and transaction ID; the only way a previous leader loses is if it is unavailable during the election.

Cluster Limplock

We find two scenarios that lead to cluster limplock. First, *a slow leader causes cluster limplock with respect to update operations.* As described above, this is because all update operations involve the leader.

Second, *the presence of a slow follower in a quorum-based protocol can create a backlog at the leader which can cause cluster limplock.* This is an interesting "hidden" backlog scenario. In our discussion above, in the presence of a slow follower, the quorum-based protocol "looks" limplock free (Z4). However,

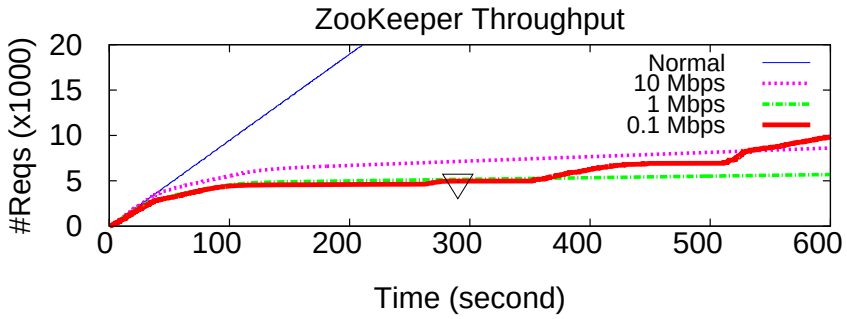

Figure 5.4: **ZooKeeper Cluster Limplock.** *The figure plots the number of requests served over time under different limpware scenarios.*

our white-box metrics hint an upcoming problem. Specifically, we monitor each queue that the leader maintains for each follower for forwarding updates, and we observe that the request queue for the slow follower keeps growing (a backlogged queue). In a short-running experiment (*e.g.*, 30 seconds in Z4), response time is not affected. However, in a larger and longer experiment, we start noticing cluster degradation (Z5).

A slow follower can cripple an entire ZooKeeper cluster. To illustrate this issue further, we plot the number of updates served over time in Figure 5.4. In a normal scenario, the throughput is constant at 90 requests/sec. With a slow follower (0.1 Mbps NIC), after 100 seconds, the cluster throughput collapses to 3 requests/sec. The root cause is resource exhaustion; the backlogged queue starts to exhaust the heap, and thus Java garbage collection (GC) works hard all the time to find free space. What happens next depends on the request and degradation rates. The leader can be in limplock for a long time (dashed lines for 10 and 1 Mbps), or it can crash as it runs out of memory after a certain time ($\bigtriangledown$ on bold line). Even after a new leader is elected, the cluster throughput is never back to normal, only 13 requests/sec (bold line after $\bigtriangledown$). This is because the slow follower

is still part of the ensemble, which means the new leader must send a big snapshot of backlog to the slow follower that can fail in the middle due to congestion and repeat continuously.

5.3.4 Cassandra

Cassandra is a distributed key-value store that partitions data across a ring of nodes using consistent hashing. Key gets/puts can be performed with consistency level one, quorum, and all. The common replication factor for a key is three. Given a key operation, a client directly connects to one of the three replica nodes (*i.e.*, the coordinator). Depending on the consistency level, a coordinator node may forward reads/writes to other replica nodes. Each Cassandra node monitors all the nodes in the cluster with dead/up labels. If a replica node is dead, a coordinator stores the updates to its local disk as "hints", which will be forwarded later when the node comes back up (*i.e.*, eventual consistency).

In our experiments, the client connects to a healthy coordinator, which then forwards requests to other replica nodes where one of them has a degraded NIC. At this point, we only analyze get and put protocols. Based on our initial results, Cassandra's architecture is in general limplock free, and only exhibits 2x slow-down. It is possible to craft more benchmarks to unearth any possible limplock cases.

Limplock Free

For weak consistency operations ("quorum" and "one"), Cassandra's architecture is limplock free (Figure 5.1, C1). However, we observe that they are not completely unaffected; when a replica node limps at 1 and 0.1 Mbps, the client

response time increases by almost 2x. We suspect some backlog/memory exhaustion similar to the ZooKeeper case, but our white-box monitoring finds none. This is because a coordinator writes outstanding requests as hints and discards them after no response for 10 seconds. We believe Cassandra employs this backlog prevention due to an incident in the past where a backlogged queue led to overflows in other nodes' queues, crippling nodes communication [51].

After further diagnosis, we find that the 2x slowdown is due to "flapping", a condition where peers see the slow node dead and up continuously as the node's gossip messages are buried in congestion. Due to flapping, the coordinator's write stage continuously stores and forwards hints. This flapping-induced background work leads to extra work by Java GC, which is the cause of 2x slowdown.

Operation Limplock

Gets and puts with full consistency are affected by a slow replica. This is expected as a direct implication of full consistency. However, in Cassandra, limplocked operations do not affect limplock-free operations, and thus Cassandra does not exhibit node limplock such as in Hadoop (Section 5.3.1) and HDFS (Section 5.3.2). This robustness comes from the staged event-driven architecture (SEDA) [176] (*i.e.*, there is no resource exhaustion due to multi-purpose threads). Specifically, Cassandra decouples read and write stages. Therefore, limplocked writes only exhaust the thread pool in the write stage, and limplock-free reads are not affected (C3), and vice versa. The SEDA architecture proves to be robust in this particular case.

5.3.5 HBase

HBase is a distributed key-value store with a different architecture than Cassandra. While Cassandra directly manages data replication, HBase leverages HDFS for managing replicas (another level of indirection). HBase manages tables, partitioned into row ranges. Each row range is called a *region*, which is the unit for distribution and load balancing. HBase has two types of nodes: *region servers*, each serves one or more regions, and *master servers*, which assign regions to region servers. This mapping is stored in two special catalog tables, ROOT and META.

We evaluate HBase by injecting a degraded NIC on a region server. In addition, as HBase relies on HDFS, we also reproduce HDFS read/write limplock (Section 5.3.2) and analyze its impact on HBase.

Operation Limplock

HDFS read/write limplock directly affects HBase protocols. All HBase protocols that perform HDFS writes (such as commit-log updates, table compaction, table splitting) are directly affected (*e.g.*, Figure 5.1, B4). The impact of HDFS read limplock is only observed if the data is not in HBase caches.

Node Limplock

An HBase region server can exhibit node limplock due to resource exhaustion by limplocked HDFS writes. The issues of fixed resource pool and multi-purpose threads also occur in HBase. In particular, a region server exhibits a node limplock when its threads are all occupied by limplocked HDFS writes. As a result, incoming reads that could be served from in-memory are affected, 620x slower in one experiment (B4).

A slow region server is a performance SPOF. Indirection (*e.g.*, HBase on HDFS) simplifies system management, but could lead to a side effect, a performance SPOF. That is, if a region server has a slow NIC, then all accesses to the regions that it manages will be in limplock (B1-3). Although a region is replicated three times in HDFS, access to any of the replicas must go through the region server. In contrast, Cassandra (without indirection) allows clients to connect directly to any replica. Regions will be migrated only if the managing server is dead, but not if it is slow.

A slow region server can lead to compaction backlog that requires manual handling. A periodic compaction job reads "sstables" of a table from HDFS, merges them, and writes a new sstable to HDFS (B5). A slow region server is unable to compact many sstables on time (a backlog). Even if the degraded NIC is replaced, the region server must perform major compactions of many sstables, which might not fit in memory (OOM), leading to a server outage. Compaction OOM is often reported and requires manual handling by administrators [3].

Cluster Limplock

A slow region server that manages catalog tables can introduce cluster limplock. Although HBase is a decentralized system, a slow region server that manages catalog regions (*e.g.*, ROOT and META tables) can introduce cluster limplock, which will impact new requests that have not cached catalog metadata.

5.3.6 Summary of Results

Our results show that impacts of limpware cascade. Specifically, operation limplock can spread to node and eventually cluster limplock. Moreover, almost all cloud systems we analyze are susceptible to limplock. Below we summarize our high-level findings and the lessons learned.

- **Hadoop:** Speculative execution, the heart of Hadoop's tail-tolerant strategy, has three loopholes that can lead to map/reduce operation limplock (Section 5.3.1). This combined with bounded map/reduce slots cause resource exhaustion that leads to node and cluster limplock where job throughput collapses by orders of magnitude. We find three design deficiencies in Hadoop. First, intra-job speculation has a flaw; if all tasks are slow due to limpware, then there is "no" straggler. Second, there is an imprecise accounting; a slow map node affects reducers' progress scores. Finally, a backup task does not always "cut the tail" as it can involve the same limpware (*e.g.*, due to memoryless retry).

- **HDFS:** Many HDFS protocols can exhibit limplock. Read/write limplock affects upper layers such as Hadoop and HBase. Block regeneration limplock could heavily degrade MTTR and MTTDL as recovery slows down by orders of magnitude. Master-slave architecture is highly prone to cluster limplock if the master exhibits node limplock. We conclude several deficiencies in HDFS system designs: coarse-grained timeouts, multi-purpose threads (lead to resource exhaustion), memoryless and revokeless retries, and timeoutless protocols. All of these must be fixed as upper layers expect performance reliability from HDFS.

- **ZooKeeper:** Our surprising finding here is that a quorum-based protocol is not always immune to performance failure. A slow follower can create a backlog of updates at the leader, which can trigger heavy GC process and eventually OOM. In this leader-follower architecture, a leader limplock becomes cluster limplock.

- **Cassandra:** Limpware does not heavily affect Cassandra. Weak consistency operations (*e.g.*, quorum) are not heavily affected because of the relaxed eventual consistency; long outstanding requests are converted as local hints. However, Cassandra is not completely immune to limpware; a slow node can lead to flapping which can introduce 2x slowdown. We also find that the SEDA architecture [176] in Cassandra can prevent node limplock as stages are isolated. More benchmarks could be crafted to unearth any possible limplock cases.

- **HBase:** Operation, node, and cluster limplocks occur in HBase similar to other systems. A new finding here is an impact of indirection (HBase on HDFS). If a region server is in limplock, then all accesses to the regions that it manages will be in limplock. Indirection simplifies system management, but could lead to a performance SPOF.

5.4 Conclusion

In this chapter, we show that limpware is a reality and a destructive failure mode; yet, it is overlooked in current large-scale cloud systems. To unearth design deficiencies in handling limpware, we present an in-depth study that quantifies the impact of limpware on these systems. Our findings show that although today's cloud systems utilize redundant resources, they are not capable of making limpware *fail in place*. Specifically, limpware can lead to limplock at many levels (operation, node, and cluster). Performance failures cascade, productivity is reduced, resources are underutilized, and energy is wasted. Therefore, we advocate that limpware should be considered as an important failure mode that future cloud systems should manage.

Chapter 6

Limplock Detection with Performance Model Checking

In the last chapter, we show that limpware can lead to limplock: a situation where the entire system progresses slowly without failing over to other healthy components. However, our approach to catch limplock bugs has two limitations. First, it is time consuming: for every protocol, we need to create a microbenchmark, set up the experiment, run the benchmark, and analyze the results manually. Second, we are unsure whether any more limplock bugs could be caught. For instance, although we only found the three bugs in Hadoop with the limpbench approach, our study of over 5,000 issues that have been reported by the Hadoop developers indicates that limplock issues are much more pervasive. This observation leads us to raise an important question: *how can we find (ideally all) limplock problems in complex systems*? To answer this important question, in this chapter, we present PMC (Performance Model-Checking), a new approach that leverages model checking to find deep limplock bugs in current cloud systems efficiently.

Model checking [111] is a verification technique that has been used to catch hard-to-find *correctness* bugs in concurrent and distributed systems [68, 82, 133, 185, 184, 183]. Its biggest challenges are state space explosion and accurate modeling. In PMC, we address these problems by leveraging *abstraction* and the *limplock property*. Employing abstraction, we create abstract models that are tractable. Focusing on the limplock property (as opposed to correctness property), we turn accurate performance modeling problem into a *relative* one, hence further simplifying the model. In our preliminary experience, we apply the PMC approach to catch limplock bugs in the Hadoop speculative execution protocol and show that PMC is a promising approach. It is possible to apply PMC to other systems such as HDFS, Zookeeper, and Cassandra as well.

This chapter is structured as follows. Section 6.1 presents our study of Hadoop bug/issue reports. Section 6.2 presents a brief description of model checking. Section 6.3 describes our approach. Section 6.4 presents our preliminary experience with using PMC to find limplock bugs in the Hadoop speculative execution protocol. Section 6.5 discusses future challenges and Section 6.6 concludes.

6.1 Study of Hadoop Bug/Issue Report

In our previous work (Chapter 5), we found only three limplock bugs in Hadoop. To estimate how pervasive this problem is, we performed a study of Hadoop bug/issue reports [7]. We scanned more than 5,000 issues over the last 5 years and studied the ones that relate to limplock problems. We did not include in our study limplock issues that happen because of many features that are not in place. For instance, in old versions of Hadoop, admission control and job preemption

Recovery Problems	Count	Description / Examples
Absent	26	Recovery is not implemented.
Incorrect	7	Recovery is executed but incorrect.
Not triggered	5	Recovery is not triggered.
Implications	**Count**	**Description / Examples**
Limplock	38	Systems progress slowly - no failover.
Dead Server	10	Servers are incorrectly marked dead.
Resource exhaustion	4	Out of resource as limplock cascades.

Table 6.1: **Limplock Problems in Hadoop.** *This table shows the results of bug/issue study in Hadoop from 2009 to 2014. In total, we found 38 limplock issues.*

features were not implemented. Thus, we do not consider limplock issues caused by the lack of these features as bugs.

In total, we found 38 limplock issues that describe the system's inability to deal with performance variability caused by degraded hardware. We present some interesting findings resulting from analyzing these issues. First, limplock issues are more widespread than we have thought. In particular, the 38 issues we found appear consistently throughout the development and deployment of many versions of Hadoop, causing severe performance problems. Although designed to be straggler-tolerant, Hadoop is still not limplock free.

Moreover, as we analyze the root cause of these issues further, we found that the straggler-tolerant recovery mechanisms of Hadoop do not anticipate certain *corner cases*. In other scenarios, the recovery mechanisms either incorrectly execute or do not execute at all, due to incorrect failure detection. Table 6.1 summarizes these problems together with their implications in handling different corner cases from real deployment. These cases are related to various factors such as failures, network topology, and job characteristics. We discuss these factors next.

- **Failures:** We found that various component failure modes such as limpware, fail-stop failures, and intermittent timeouts can make the system be in limplock. Moreover, failures can happen at different points during the execution of a system's protocol. For instance, slowdown can happen not only during the task execution phase but also during the preparation phase when binary code is transferred to the machines where the tasks will run [50]. As another example, slowdown can happen after all the tasks are close to finishing [49]. Thus, an efficient testing framework must be able to exercise various failure modes at different points in the system protocol.

- **Topology:** Interestingly, we found that Hadoop recovery mechanisms are problematic in certain network topologies. Hadoop backlisting protocol is an example. In this protocol, reduce tasks can blacklist map nodes with too many shuffling errors. This protocol works just fine for a single-rack topology, but is problematic in multi-rack settings where connection bandwidth between racks is significantly degraded. Specifically, it is often "wrong", and "counterproductive", according to the Hadoop developers:

  > *"In cases where networking problems cause substantial degradation in communication across sets of nodes - then large number of nodes can become blacklisted as a result of this protocol. The blacklisting is often wrong (reducers on the smaller side of the network partition can collectively cause nodes on the larger network partitioned to be blacklisted) and counterproductive (re-running maps puts further load on the (already) maxed out network links)."* [52].

- **Job placement:** Finally, we found that nondeterminism in runtime environment can lead to different job placement scenarios (*e.g.*, where a task is scheduled). This can make Hadoop detect failures incorrectly, thus do not trigger its recovery mechanisms timely and correctly. For instance, we found a case where speculative execution was not triggered due to a corner case in job placement [44].

In summary, our study shows that limplock issues are more pervasive than we have thought. Moreover, all the issues arise because Hadoop does not anticipate various deployment corner cases such as different failures, network topologies, and job placements. This observation strongly motivates the need for a new approach that is more effective and efficient in catching limplock bugs.

6.2 Model Checking

Model checking is a process that drives the system to all possible states in order to find intricate corner cases that adversely impact the system. This process is controlled by a model checker. Given a system model, the model checker starts from an initial state, recursively generates successive states by exploring *nondeterministic* events in the system, and checks if the property being verified holds. A *state* of the system is a snapshot of the entire system at certain execution point; the *state space* of the system is the set of all system states reachable from the initial state. The model checking process terminates when the state space is completely explored or when the model checker runs out of resources.

The main advantage of model checking is that it is automatic and extensive. Given a system model and a property that needs to be verified, the model checker

automatically explores the state space. If the property is violated at a certain state, the model checker can produce the execution sequence leading to that state, making it easier for the user to debug the problem. Moreover, since the state space is finite, the model checker can, in principle, exhaustively explore the entire state space. As a result, it can find *all* deep corner cases at which the property of interest does not hold.

One of the biggest drawbacks with using model checking is the problem of state space explosion. That is, the state space can grow exponentially as the model gets bigger and more complicated. Thus, resources (*e.g.*, memory) required by a model checker can quickly grow beyond the capacity of modern machines. Moreover, *accurately* modeling a system, especially its performance aspect, is challenging [36, 115].

In this chapter, we address these problems by using *abstraction* and leveraging the *limplock property*. Abstraction is a well-known technique that has been used in model checking to simplify the system models, making model checking feasible [118]. Using abstraction, we can construct a model that captures only details (*e.g.*, protocols and data structures) that matter, thus significantly simplifying the model. Moreover, by leveraging the limplock property, we do not need accurate performance modeling, as limplock bugs can be caught using *relative* performance comparison. We present our approach in detail in the next section.

6.3 Our Approach

We present PMC (Performance Model-Checking), a new approach to catch limplock bugs in complex systems. As in the traditional model checking approach, we

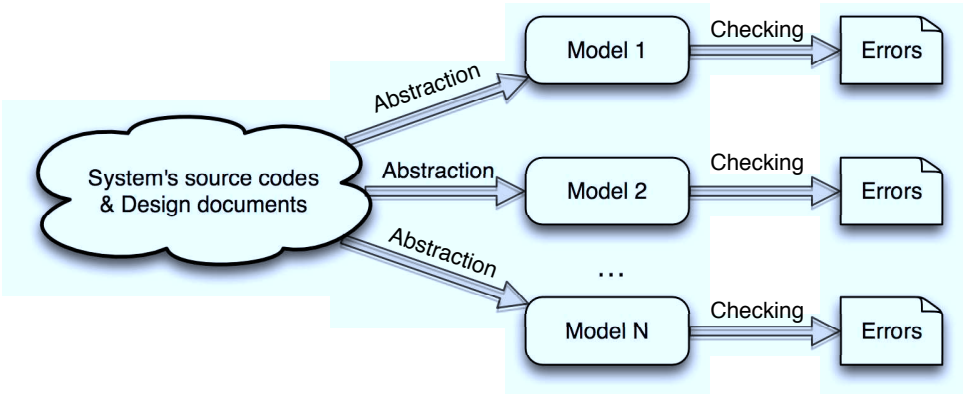

Figure 6.1: **Overview of PMC.** *This figure presents an overview of PMC. Starting from the system source code and design documents, we create different abstract models for different protocols and individually model check them.*

first model the systems of interest using a modeling language (we use Petri nets in our approach). We focus on data intensive protocols of the systems because they are potentially being affected by limpware. By analyzing the system source code and design documents, we abstract out system-level constructs (*e.g.*, threads, queues, timeouts, and locks) to create models that are able to depict in detail how system components interact under performance failure. Given the abstract models, we then model check each of them to find potential limplock bugs.

6.3.1 Extracting Abstract Models

Different from traditional approaches, our approach does not create one detailed model for a given system. Instead, given a complex system, potentially with many protocols to verify, we create different abstract models for different protocols and individually model check each of them (Figure 6.1). The key factor in our approach is abstraction, a technique that eliminates unnecessary details to simplify the resulting model and make it tractable. To construct an abstract model for a par-

ticular protocol, we abstract away details that are irrelevant to the protocol being analyzed and the property of interest, making the model smaller than the detailed one. Specifically, we model relative performance, replace a data structure with a simpler one, and eliminate unrelated data structures.

- **Modeling relative performance:** As our ultimate goal is to catch limplock bugs, we need to model the performance aspect of all the actions involved in a protocol. For instance, we need to specify the latency of an action (*e.g.*, a disk access) or the throughput of a degraded disk or NIC. However, *accurate* performance modeling is challenging: it is hard to answer questions such as what is the peak throughput of the system under certain workload, given that there is 1-Kbps NIC in the system? Fortunately, limplock can be detected using *relative* performance comparison. Given a piece of limpware in the system, we only need to find out if the overall performance of the system is significantly affected, compared with the no failure case. As a result, we specify relative performance in our abstract model. For instance, we can specify that a healthy NIC takes `X` units of time to transfer a message, while a limping NIC takes `1000X` units of time to perform similar task. With relative performance abstraction, our model is simplified substantially.

- **Replacing data structures:** Replacing complex data structures in the original system with simpler ones can also reduce the amount of state in the model. For instance, in Hadoop, map and reduce code can handle different data formats. However, as the content of actual data makes little difference in the performance of the map and reduce code, the complex data type is replaced with a simpler one in the abstract model. Another simplification

that can reduce the state space is limiting the range of a data type. For instance, instead of specifying a latency variable to have any possible value in the range `[0, 1000]`, we can define an enumeration type with limited number of values (*e.g.*, `0`, `10`, and `1000`) for which the variable could bind to in different scenarios (*e.g.*, limpware vs. no failure). Again, we can do this safely because we do not need accurate performance modeling to catch limplock bugs.

- **Eliminating data structures:** Some data structures can be eliminated completely, without affecting the validity of the resulting abstract model. For example, Hadoop maintains history information for finished jobs, which is not relevant to the limplock property and can safely be eliminated.

6.3.2 Performance Model-Checking

In traditional model checking, given a model, the model checker starts from initial state, explores nondeterministic events to construct the state space, and performs checks to unearth correctness problems such as potential deadlock and data loss [68, 82, 133, 183, 184, 185]. Unlike these previous works, PMC focuses on performance problems. Specifically, during state space exploration, instead of checking for correctness properties, the PMC model checker checks for the *limplock property* (*i.e.*, if the system is in a limplock scenario). Given an abstract model written in a modeling language, we perform the following steps to find limplock bugs.

First, we need to specify the initial state of the model. For instance, we specify the number of nodes involved in the protocol, the workload (*e.g.*, the number of

tasks to executed), and the number of failures to inject. Second, we specify the nondeterminism in the system. This allows the model checker to explore multiple transitions from a single state. Which nondeterminism to specify depends on specific protocols. For instance, in Hadoop, we can specify which node is experiencing limpware among all the nodes in the system, which phase in the job execution process the fault happens, where a data block is stored, and where a task is scheduled.

Finally, we provide checks to verify the limplock property of the protocol. That is, we want to check if the system suffers from limplock at a certain state. The checks are evaluated at each state during the state space exploration. For instance, one can write a simple check to test if the execution time of a job exceeds a particular threshold, which can be obtained by executing the model without any failure. Here, we leverage relative performance comparison to write the limplock check.

6.3.3 Petri Net

We use Petri nets, a mathematical modeling language, to describe distributed systems (we could use other modeling languages as well). Being around for decades, Petri nets have been used to model and analyze various systems, from network protocols to workflow systems [74, 169].

The classical Petri net, invented by Carl Adam [142], only supports the modeling of states, events, conditions, choice, iteration, synchronization, and parallelism. However, it does not support the modeling of data, time, and hierarchy. Recent extensions support these features, and thus facilitate the modeling of complex systems where data and time are important. In our approach, we use Colored Petri

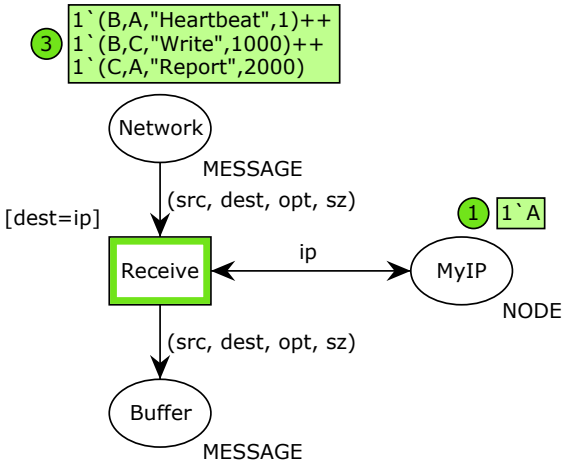

Figure 6.2: **Simple NIC model.** *A Petri net model of a machine network card that receives messages from network to its internal buffer.*

Nets (CPN) [109], a framework that supports all of these extensions. With CPN, high-level system concepts such as processes, communication channel, synchronization primitives, timeout, and retries can be modeled easily [109, 116, 174].

To verify our abstract model, we use CPNTool, a model checker that supports state space calculation on a Petri net model [109]. It provides two state space exploration modes: simulation and exhaustive. In *simulation* mode, single execution sequence is explored; random choices are made between consecutive states. We use the simulation mode to obtain performance metrics of the system in the no failure cases. In *exhaustive* mode, the entire state space is explored. Moreover, users can provide additional predicates to control the state space exploration process. The exhaustive mode is used to check our models for limplock bugs.

Example: Modeling a Network Interface Card

To illustrate the basic syntax of Petri nets and its ability, we present a simple example of a Petri net model (Figure 6.2). Here, we model a machine network card (NIC) that simply receives messages from the network to its internal buffer. We also model its performance aspect, *i.e.*, how long it takes for the *receive* action.
State: Petri net syntax has various constructs to model *states* and *actions* of a system. States are represented by *places*, drawn as ellipses or circles. The NIC is modeled using three different places: `Network`, `Buffer`, and `MyIP`. The place `Network` models the communication channel among the nodes in a network. The places `MyIP` and `Buffer` model the IP address and internal buffer for receiving messages, respectively. Each place has a certain *type (color set)* to represent the data structure that the place can contain. Types can be primitive (*e.g.*, real, int, and string) or complex (*e.g.*, a record). The type definitions used in Figure 6.2 are as follows:

```
// Type definitions
colset NODE = with A | B | C;
colset OPERATION = string;
colset SIZE = int;
colset MESSAGE = product NODE * NODE * OPERATION * SIZE;
```

The type `NODE` is defined as an enumeration containing three possible values representing IP addresses of all machines in a cluster. Here, we model a cluster with only three nodes `A`, `B`, and `C`. The types `OPERATION` and `SIZE` are defined as string and int respectively. Finally, the type `MESSAGE` is defined as a record of four fields representing source, destination, operation, and size of a message.

A certain state of a Petri net model is represented as a *marking*, consisting of

a number of *tokens* on individual places. Each token represents a *value (color)* belonging to the type of the place where the token resides. For instance, in Figure 6.2, the place `MyIP` has a single token with value `A`, representing that the NIC belongs to machine `A`. The place `Network` has three tokens representing three messages currently floating around the network.

Action: Actions of a system are modeled as *transitions*, drawn as rectangles. Transitions and places are connected by *arcs*. When an action occurs, state (*i.e.*, marking) of the model is changed: tokens from places connected to the incoming arcs of the transition are removed, and new tokens are added to output places connected to the outgoing arcs. Which tokens and how many of them are removed and added every time a transition occurs are determined by *arc expressions*, written next to the arcs. In our example, the NIC has only one transition `Receive` that models the action taken when a message is transferred from the network to the machine's internal buffer. When a message is received, the transition removes a token from the place `Network` and adds the same token to the place `Buffer`.

Dynamic behaviors: We now describe the conditions under which a transition may occur. For a transition to be *enabled* (*i.e.*, ready to occur), it must be possible to *bind* data values from the transition's input places to the *variables* appearing in the arc expressions. In this case, the transition is said to be *color enabled*. For instance, the transition `Receive` has five variables declared as:

```
// Variable declarations
var src, dest, ip: NODE;
var opt: OPERATION;
var sz: SIZE;
```

A possible binding of the transition `Receive` is as follows, where ":-" should be

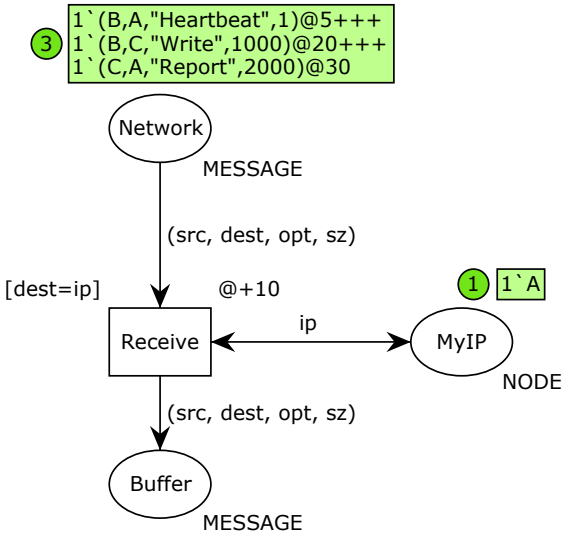

Figure 6.3: **A NIC model with time.** *Tokens now have timestamps and transitions now have latency inscription.*

read as “bound to”:

```
 src :- B
dest :- A
 opt :- ‘‘Heartbeat’’
  sz :- 1
  ip :- A
```

It is also possible to attach a *guard*, a Boolean expression, to a transition in order to control its behavior. The guard specifies that the transition only accepts bindings for which the Boolean expression evaluates to be true. In our example, the transition `Receive` has a guard `dest = ip`, specifying that the NIC only receives messages sent to the machine the NIC belongs to.

Timing: We model the performance aspect of a system using the time concept of Petri nets, which is supported by the introduction of a *global clock*. The clock

values represent *model time*; each token can carry a *timestamp*; each transition can be associated with a *transition latency value*. With timing, a transition may occur under two conditions. First, it must be *color enabled*: the required tokens are presented at the transition's input places and the guard (if any) must be evaluated to be true. Second, the transition must be *ready*, *i.e.*, the timestamps of required tokens at its input places must be less than or equal to the current model time.

Figure 6.3 shows a timed version of the NIC model; its structure is similar to that of the untimed model in Figure 6.2, with two differences. First, the tokens at the place `Network` now have timestamps to model the message arrival times. Second, the transition `Receive` is augmented with an inscription `@10` that models how long the receive activity takes (in this case, `10` units of time). This means that the timestamps of the output tokens are `10` time units larger than the clock value at which the transition occurs.

6.4 Case Study

We now present a case study in which we apply PMC to find limplock bugs in the Hadoop speculative execution protocol. Our study of Hadoop bug/issue reports shows that Hadoop is still limpware-intolerant, hence additional verification for Hadoop is useful. First, we describe our model of Hadoop in Petri nets. Then, we present the model checking results showing that PMC is more effective than our previous approach.

6.4.1 Model checking Hadoop with PMC

The Hadoop source code is complex with 50,000 line of Java code, making it challenging (if not impossible) to model the full Hadoop detail. However, using abstraction, we focus on speculative execution, a specific protocol of Hadoop, and eliminate irrelevant parts to create a verifiable abstract model. The Hadoop speculative execution protocol involves three major components: client, JobTracker, and TaskTracker. The client submits new jobs that contain many map and reduce tasks to the JobTracker, which then schedules the tasks to multiple TaskTrackers. The JobTracker monitors status of running jobs and answers queries for job status from the clients. If a task instance is slow in progress, the JobTracker launches a backup instance for the same task (*i.e.*, speculative execution).

We leverage our abstraction technique to abstract out irrelevant details when modeling the speculative execution protocol. The full model of the protocol is presented in Appendix B. Here, we briefly describe a simplified version of the JobTracker logic. Figure 6.4 presents simplified source code of the JobTracker, which is essentially an RPC server that communicates with the clients and TaskTrackers through RPC messages. To simplify the JobTracker model, we eliminate parts that relate to the client interactions (*e.g.*, job submission) and model only the TaskTracker interactions (*e.g.*, job scheduling and failure handling). Specifically, we model how the JobTracker handles two RPC messages, `heartbeat` and `getMapComplete`, which are the key operations in speculative execution.

The JobTracker model is illustrated in Figure 6.5; the color sets (data types) used in the model are defined Figure 6.6. In the model, RPC requests and responses are modeled by the places `Msg_in` and `Msg_out` respectively. These two places have the type `MSGxTIME` representing network messages with time-

```
class JobTracker implements TrackerProtocol, // TaskTracker operation
  JobSubmissionProtocol, AdminProtocol     // Client operation
{
   JobTracker() {
     // ...
     // omitted code
     // ...

     // start RPC server
     server = RPC.getServer(...)
     server.start();
     // omitted code
   }

   // RPC handlers for client job submission
   JobStatus submitJob(JobID jobName) {...}
   JobStatus getJobStatus(JobID jobid) {...}
   // ...
   // > 20 client RPCs omitted



   // RPC handlers for taskTracker
   synchronized HeartbeatResponse heartbeat(TaskTrackerStatus status) {
     // omitted code
     processHeartbeat(status); // update task status
     HeartbeatResponse response = new HeartbeatResponse();
     List<TaskTrackerAction>
        actions = new ArrayList<TaskTrackerAction>();
     // omited code
     List<Task> tasks = taskScheduler.assignTasks(status.getName()));
     for (Task task: tasks)
        actions.add(new LaunchTask(task));
     response.setActions(actions);
     return response;
   }

   TaskCompletionEvent getMapCompletion(TaskID taskid) {...}

   // omitted code
}
```

Figure 6.4: **JobTracker Source Code.** *The Job Tracker is an RPC server that handles requests from clients for job submission and TaskTrackers for job scheduling. Code is simplified for readability.*

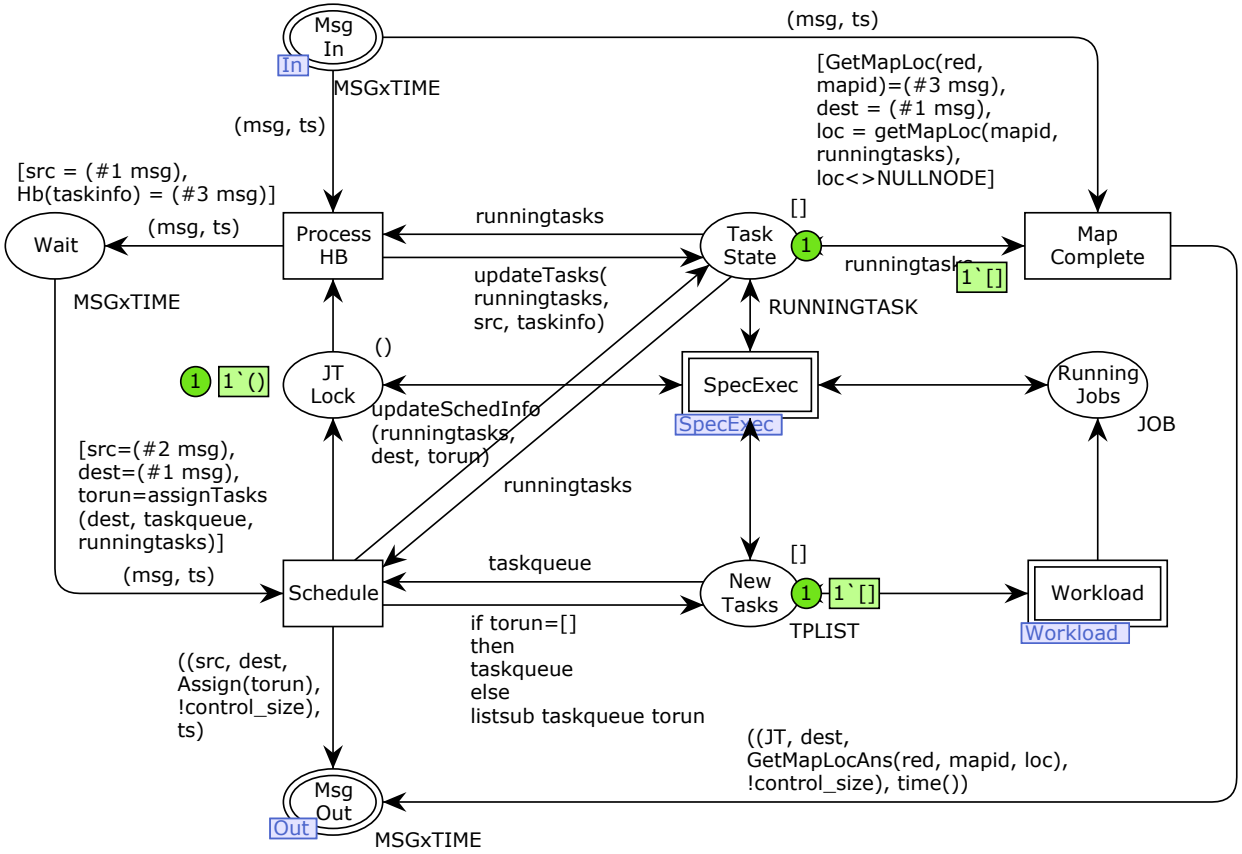

Figure 6.5: **The JobTracker Model.** *This figure represents the Petri net that models the logic of the JobTracker in detail.*

```
colset NODE = with JT | A | B | C | D | NULLNODE | R12 | R21;
colset NODELIST = list NODE;
colset TASKTYPE = with M | R; (* Map/Reduce *)
colset JOB = product STRING * INT * INT;
colset NUM = INT;
colset ATTEMPT = INT;
colset PARAMS = NODELIST;
colset PROGRESS = INT; (* Progress score *)
colset TASKID = product JOB * TASKTYPE * NUM;
colset TASK = product JOB * TASKTYPE * NUM * ATTEMPT;
colset TASKxPARAMS = product TASK * PARAMS;
colset RUNNINGTASK = list NODExTASKxPARAMSxPROGRESSxTIME;
```

Figure 6.6: **Color Set Definitions for the JobTracker Model.** *In the figure, "colset" is a keyword used for declaring new types, "with" for an enumeration type, and "product" for a record type.*

stamps. The place `Task_State` (with type `RUNNINGTASK`) models the state of all scheduled tasks (*e.g.*, running, failed, pending, and complete) and their related information such as task progress, start time, and complete time. When processing heartbeats from TaskTrackers, the JobTracker updates the progress of all running tasks (transition `Process_HB`) and replies back with new tasks (including speculative tasks) to be run on the TaskTrackers (transition `Schedule`). Speculative tasks are calculated by the transition `SpecExec`. The JobTracker also handles `getMapComplete` messages from reducers asking for the location of completed map tasks (transition `MapComplete`). The global lock of the JobTracker process is modeled by the place `JT_Lock`. Finally, instead of having a part representing client interactions, the model has a `Workload` module to generate new jobs.

To model certain deterministic and atomic computation such as updating task status, finding speculative tasks, and assigning tasks, we have two choices. We could either emulate the computation completely in Petri net structures or represent the computation with high-level functions that can be embedded in the Petri net model. We choose the latter approach to keep the model simple and reduce

its state space. Specifically, we convert certain computations from Hadoop source code to ML functions and embed them into the Petri net model. For instance, in the JobTracker model, we put the ML functions such as `assignTasks` and `updateTasks` in the guards of transitions `Process_HB` and `Schedule`. As the computations are deterministic and atomic, "pruning" the Petri net this way does not lose the opportunity to introduce nondeterminism into the model.

To further reduce the model's state space, we limit the domain of certain data types to minimize the number of branches when a binding decision is made. For instance, instead of specifying a latency variable to have any value in the range `[0, 1000]`, we define a type with a limited number of values (*e.g.*, `0`, `10`, and `1000`) to which the variable could bind. We also limit the number of nodes and tasks in the system. Finally, we convert *infinite* periodical activities (*e.g.*, send a heartbeat every 10 units of time) into *finite* ones (*e.g.*, periodically send heartbeats for 20 times then stop). These abstractions make the model checking process feasible by significantly reducing the state space. However, some limplock corner cases might be missed; for example, if limplock only happens in a cluster with a large number of nodes, limiting the number of nodes may make the model checker miss the bug.

Given the Hadoop model, the model checker starts from an initial state, recursively generates successive states by exploring nondeterministic events in the system. Nondeterminism comes from three primary sources: job characteristics, scheduling, and fault injection. In our model, we vary input and output data locations for map and reduce tasks respectively. Regarding scheduling, because the JobTracker assigns new tasks to a TaskTracker when processing its heartbeat, we make the TaskTrackers send heartbeat messages concurrently, thus introducing

```
(* Returns list of states where a job execution time
   is X times larger than normaltime - the average
   execution time of a job no failure case *)
fun FindLimplockStates([], job, X, normaltime) = []
  | FindLimplockStates(s::statelist, job, X, normaltime) =
let
  val taskstate = hd(Mark.JTCore'Task_State 1 s)
  val time = completeTime(job, taskstate)
in
  if time >= X * normaltime  (* relative check *)
  then s::FindLimplockStates(statelist, job, X, normaltime)
  else FindLimplockStates(statelist, job, X, normaltime)
end;
```

Figure 6.7: **Limplock Specification.** *This figure presents an example of a specification for the limplock property. This specification is checked by the model checker at every step during the state exploration.*

nondeterminism to job placement. Finally, we inject limpware (*e.g.*, a slow NIC) to a random node in the system; we leave crashes and packet loss for future work.

Since we are interested in the limplock property (*i.e.*, whether the system experiences limplock at a certain state), we provide *limplock specifications* to the model checker. These specifications are then evaluated at each state of the state exploration process. Violation happens when there is a state in which a job takes *relatively* and *significantly* longer time to execute, compared with the no-failure cases. As a result, our limplock specifications are essentially simple ML functions that take a state instance and return true if a job takes a relatively long time to finish. By leveraging the limplock property, we simplify our checks in two ways. First, our checks focus only on the places representing limplock metrics (*e.g.*, task/job execution time). Second, as performance in our model is relative, our checks contain only relative comparison. Figure 6.7 presents an example of a limplock check. The check `FindLimplockStates` has four parameters: a list of states in the state space, a job of interest, a threshold `X`, and the average execu-

tion time of a job in no-failure cases. It iterates through each state in the list, and for every state, focuses only on the place `Task_State` of the JobTracker's core module (Figure B.4) as the place models the execution information of all the jobs in the cluster. The check simply extracts the execution time of the job encoded in the token value of this place (function `completeTime`) and performs relative comparison to find out if the state experiences limplock, *i.e.*, if the job takes `X` times longer to finish, compared to the normal cases. We obtain the normal execution time of a job as a base for relative comparison by using the simulation mode of the model checker. Specifically, we run our Hadoop model *without any failure* many times and compute the average job execution time. Here, one challenge arises when the model, without any failure, has a large amount of performance variance, which potentially causes *false positives*. In our experience of running limpbench in the previous chapter, when we increase the slowdown factor of a hardware component to a large enough value (*e.g.*, 1000x slowdown), the performance difference between the true limplock case and the no-failure case is often orders of magnitude. As long as the variance is within this range, there will be no false positive. Nevertheless, care should be taken when choosing a threshold for relative comparison in order to minimize the chance of false alarms.

The full model of Hadoop is presented in Appendix B. We spend in total two days to create this model. The translation of deterministic and atomic computations to ML functions is the most time-consuming part because we were not familiar with the ML programming language before this project. However, it is possible to speed up this process by leveraging static analysis to automatically extract the model from the system source code (Section 6.5).

PMC Results	
Number of States	300,000
Time (minutes)	65
Limplock Bugs found	5

Table 6.2: **Performance Model-checking Results.** *This table presents the results of model checking the Hadoop speculative execution.*

6.4.2 Results

Table 6.2 summarizes the model checking results. The model checker runs for about an hour and produces a state space of more than 300,000 different states. By applying our limplock specifications, we found five scenarios where the Hadoop speculative execution protocol is broken under a limpware fault (*e.g.*, a slow NIC). Table 6.3 summarizes these scenarios. Interestingly, with PMC, we found two new bugs that were not found with the limpbench approach.

We now describe these bugs in detail; we use the term "slow node" to refer to the node with a degraded NIC. The first three bugs were also found using our previous approach (Section 5.3.1). Thus, we only describe the two *new* cases that the limpbench approach was not able to find.

First, *Hadoop speculative execution is not triggered, if all the map tasks of a job experience limplock (Bug 4).* Specifically, there is a corner case in which each map task of a job experiences limplock while reading its input data. Here, because of nondeterminism in task placement, data locality is not achieved. Specifically all the map tasks read their input from remote nodes. Each map task either runs in a normal node but reads data from the slow node or runs in the slow node and its input data is not local. Nevertheless, this scenario rarely happens in practice; as the number of map tasks is typically large and their input data are replicated, it is unlikely for all the map tasks to end up reading from the same slow node.

ID	Description	Limpbench	PMC
Bug 1	All reducers are slow due to slow map node	√	√
Bug 2	All reducers are slow due to writing to same slow node	√	√
Bug 3	Original and backup mappers read from same slow node	√	√
Bug 4	All mappers are slow due to reading from same slow node	×	√
Bug 5	Hadoop speculative execution is not topology-aware	×	√

Table 6.3: **Hadoop limplock bugs.** *This table describes all five bugs that are found using the PMC approach. Bugs 4 and 5 were not found using the limpbench approach presented in the last chapter.*

Perhaps, this is why we did not find this adverse corner case with limpbench. This further shows the power of our model checking approach: we can catch deep and rare corner cases that could not be found with traditional testing.

Second, *Hadoop speculative execution is not topology-aware (Bug 5).* This scenario happens when map tasks and reduce tasks are in different racks and we have a faulty network link connecting the two racks. The fault does not cause stragglers (*i.e.*, slow tasks), but instead faulty/timeout connections when a reducer wants to get some output from the map. Here, these intermittent timeouts are dealt as fail-stop failure, *i.e.*, Hadoop re-runs the same map task in another node (hoping that this will be successful because the previous problem was assumed because of a bad node). But since the map task is re-run in the same bad-map rack, the problem will repeat because it is the network link that is faulty. Here, incorrect root-cause diagnosis leads to fatal consequence. This bug is not found previously perhaps because we did not exercise different network topologies.

We also explore the feasibility of limplock anticipation by introducing the fixes for certain bugs to the model and showing that the fixes indeed work. Specifically, we introduce fixes for the limplock case where *both the original and backup map tasks can be in limplock when they read from the same remote slow node (Bug 3).* Here, the underlying issue is *infoless recovery*: the backup task could

potentially read from the same slow node as the original. We introduce a simple *info-aware* fix (*i.e.,* we model the backup task so that it avoids reading from the same node as the original) and rerun the model checker. We do not find the case for Bug 3 in this rerun (other bugs are still there). This result proves that our fix is effective in the Hadoop model.

6.5 Discussion, Limitations, and Future Work

The preliminary results from the last section show that PMC is effective. Nevertheless, there are still many open challenges, which we discuss next.

- **Automatic model extraction**. So far, in our approach, we inspect the Hadoop source code and design documentation to *manually* construct the model. This approach is useful when we first explore the feasibility of PMC, but it does not scale when we want to apply PMC to many other protocols. Fortunately, static analysis has proven to be useful in extracting models from source code for various purposes [118, 174]. In these approaches, however, target systems are often written in special languages that facilitate the extraction. In our case, we plan to apply PMC to open-source cloud systems whose code bases are not ready for automatic extraction. To address this challenge, it is possible to leverage domain-specific annotation in the extracting process.

- **Structural and behavioral checks**. Beyond the limplock property (*i.e.,* are there any cases of limplock?), we are also interested in debugging the root cause (*i.e.,* what design flaws lead to limplock?). We can answer this question by writing structural and behavioral checks. First, since limplock

often results from bad systems designs such as multi-purpose threads and queues, unbounded queues, and big-lock dependencies (Chapter 5), we can write *structural checks* that analyze Petri net structures representing system constructs to pinpoint such bad designs. Moreover, as understanding how the systems react in a limplock scenario can gain invaluable insights, we can write checks to capture their behaviors. Specifically, we can write *behavioral checks* to answer questions such as do the systems attempt to recover, what are those recovery actions, and how many times they are performed.

- **Limplock anticipation**: Model checkers are useful in finding corner case bugs. Once a limplock bug is pinpointed, we can change the old model to incorporate various recovery principles (info-aware recovery in the last section is an example). The revised model is then checked again using the same process described in the previous section to make sure the fixes are effective. It is possible to go through this cycle multiple times until we finally obtain a limplock-free model. Up to this point, we can be confident that the solution indeed works. We then apply the solution to the system's source code.

6.6 Conclusion

Limplock is widespread throughout the development and deployment of large-scale cloud systems, leading to severe performance problems; yet, it is challenging for developers to effectively pinpoint limplock bugs. To address this pressing challenge, we propose PMC, a powerful performance model-checking approach that is capable of finding subtle limplock bugs in complex distributed systems.

Similar to traditional model checking, the PMC approach is able to push the system of interest to intricate corner cases. Different from traditional model checking, the PMC approach leverages the limplock property to enable *relative* performance modeling, as opposed to accurate performance modeling, which is hard to accomplished. Moreover, PMC also leverages abstraction to eliminate irrelevant details, making the resulting models small and tractable. We apply the PMC approach to find limplock bugs in an interesting case study: the Hadoop speculative execution protocol. The full model of Hadoop is described in detail in Appendix B. Although there still many challenges to address, preliminary results show that PMC is useful and promising.

Chapter 7

Related Work

In this chapter, we discuss various research efforts that are related to this dissertation. We first discuss literature on failure exploration and system specifications. We then discuss related work on handling fail-silent failures and close this chapter with other efforts in tackling performance failures.

7.1 Failure Exploration and System Specifications

In this section, we discuss research efforts on failure exploration and system specifications, which are related to FATE and DESTINI.

Failure Exploration: Developers are accustomed to easy-to-use unit-testing frameworks. For fault-injection purposes, unit tests are severely limited; a unit test often simulates a limited number of scenarios. As a result, the code is bloated; the HDFS unit test is over 20 KLOC (almost as big as HDFS) but by no means covers the space of failure scenarios. In particular, it exercises very few scenarios with multiple failures. When it comes to injecting multiple varieties of failures,

one common practice is to inject a sequence of *random* failures as part of the unit test [57, 167].

To improve common practices, recent work has proposed more exhaustive fault-injection frameworks. For example, the authors of AFEX and LFI observe that the number of possible failure scenarios is "infinite" [113, 129]. Thus, AFEX and LFI automatically prioritize "high-impact targets" (*e.g.*, unchecked system calls, tests likely to fail). So far, they target non-distributed systems and do not address multiple failures in detail.

Recent system model-checkers have also proposed the addition of failures as part of the state exploration strategies [114, 182, 183, 184]. Modist, for example, is capable of exercising different combinations of failures (*e.g.*, crashes, network failures) [183]. However, exploring multiple failures creates a combinatorial explosion problem. This problem has not been addressed by the Modist authors, and thus they provide a random mode for exploring multiple failures. Overall, we found no work that attempts to systematically explore multiple-failure scenarios, something that cloud systems face more often than other distributed systems in the past [37, 69, 100, 105]. In this dissertation, we introduce FATE, a novel failure exploration framework, which is capable of systematically exploring multiple failure scenarios; it also solves the problem of combinatorial explosion with smart pruning strategies.

System Specifications: Failure injection addresses only half of the challenge in recovery testing: exercising recovery code. In addition, proper tests require specifications of *expected behavior* from those code paths. In the absence of such specifications, the only behaviors that can be automatically detected are those that interrupt testing (e.g. system failures). One easy way to write extra checks is to

write them as part of a unit test. Developers often take this approach, but the problem is there are many specifications to write, and if they are written in imperative languages (*e.g.*, Java) the code is bloated. For these reasons, the number of written specifications is usually small.

Some model checkers use existing consistency checks such as fsck [184], a powerful tool that contains hundreds of consistency checks. However, it has some drawbacks. First, fsck is only powerful if the system is mature enough; developers add more checks across years of development. Second, fsck is also often written in imperative languages, and thus its implementations are complex and unsurprisingly buggy [99]. Finally, fsck can be considered as "invariant-like" specifications (*i.e.*, it only checks the state of the file system, but not the *events* that lead to the state). Nevertheless, specifying recovery requires "behavioral" specifications.

Another advanced checking approach is WiDS [124, 125, 183]. As the target system runs, WiDS interposes and checks the system's internal states. However, it employs a scripting language that still requires a check to be written in tens of lines of code [124, 125]. Furthermore, their interposition mechanism might introduce another issue: the checks are built by interposing specific implementation functions, and if these functions evolve, the checks must be modified. The authors have acknowledged but not addressed this issue [124].

Frameworks for declarative specifications exist (*e.g.*, Pip [146], P2 Monitor [157]). P2 Monitor only works if the target system is written in the same language [157]. Pip facilitates declarative checks, but a check is still written in over 40 lines on average [146]. Also, these systems are not integrated with a failure service, and thus cannot thoroughly test recovery.

Overall, we found no framework that enables developers to write clear and

concise recovery specifications for real-world implementations of today's cloud systems. Existing work use approaches that could result in big implementations of the specifications. Managing hundreds of them becomes complicated, and they must also evolve as the system evolves. Thus, in practice, developers are reluctant to invest in writing specifications [10] – hence the number of written specifications is typically small and does not scale to the complexity of the system. We advanced the state-of-the-art by introducing DESTINI, a framework that enables developers to write hundreds of specifications clearly, concisely, and precisely.

7.2 Handling Fail-silent Failures

In this section we discuss related work on which HARDFS is based and other approaches that address memory corruption and software bugs.

HARDFS is primarily based on two related works: N-version programming (NVP) [29] and Micro-reboot [54]. To detect fail-silent behaviors, an NVP-based system uses N different versions with separate internal state and implementation logic. Although appealing, NVP has two major challenges that make it difficult to apply in practice. First, the engineering effort required to develop multiple versions of a software system is high. Second, even if different versions are available, coordinating them is nontrivial. For instance, an N-version file system such as EnvyFS [32] requires complex machinery to coordinate different file system implementations; it also incurs significant overhead as it must issue operations to at least two different file systems. HARDFS reduces engineering costs by only protecting select subsystems with redundant implementations. HARDFS minimizes overhead by making data structures lightweight via lossy compression.

Lossy compression occasionally causes unnecessary recovery due to error-detection false positives; we make this acceptable by making recovery inexpensive with Micro-reboot, which advocates that systems should be designed with the ability to reboot partial components [54]. Micro-reboot has been useful in other systems, allowing OS drivers and file systems to be restarted without a full OS reboot [163, 164, 165].

A common way to address memory corruption (but not most bugs), is to add detection machinery at the hardware and software layer (*e.g.*, using ECC memory and page checksums). These approaches do not protect the system from bugs introduced by complex software in many layers (*e.g.*, firmware, OS, and the application itself). An end-to-end approach to handling corruption is provided by PASC [67], a library that makes it easy for developers to maintain two replicas of the main state and execute the program logic twice on both replicas. PASC involves minimal engineering effort since developers do not need to implement the same functionality twice; however, simply executing the same code twice makes the system vulnerable to bugs in that code. Furthermore, keeping two complete state replicas is costly.

One way to address bugs (but not memory corruption) is to perform offline testing driven by sophisticated model checkers [96, 97, 183, 184]. Model checking is complementary to SLEEVE. It is more desirable to find and fix a bug during testing than to tolerate the bug during deployment; however, offline testing can only address bugs that arise in the situations selected by the model checker's execution-exploration and state-exploration algorithms. By contrast, SLEEVE performs checking in every situation that arises in deployment.

Some systems, like HARDFS, attempt to address both bugs and memory cor-

ruption. For example, Recon [86] interposes on all disk writes by the file system, and prevents any writes that would break *fsck*'s consistency rules. Although relatively lightweight, Recon only checks for consistency, not correctness in general. Byzantine Fault Tolerance (BFT) [55, 121, 147] is a heavy-weight solution which protects software systems from malicious behaviors like corruption, bad inputs, and wrong computation. Unfortunately, BFT requires a high degree of replication ($3f + 1$ replicas to tolerate f failures), does not handle cases where the logic of the software is buggy, and may be difficult to deploy (*e.g.*, requires significant changes to the HDFS replication policies [63]).

7.3 Handling Performance Failures

This section provides the connections and distinctions between our work on the impact of limpware analysis and recent hot topics in systems.

The concept of hardware performance failure has been introduced before [27] and its impact was first studied in the context parallel algorithms such as sorting [24]. We are the first to study the impact of limpware in the context existing large-scale cloud systems.

Recent work provides rich analysis of various hardware failures including machine failures, disk failures, memory corruption, and network failures [30, 85, 89, 150, 151, 170]. Formal studies of these failures were undertaken after anecdotes started to circulate. Based on our analysis, we argue that similar studies of limpware are needed.

Distributed jobs have to deal with performance variability originating from jitters and stragglers. Jitters are often transient and sporadic in nature [190]. Limp-

ware on the other hand can be both transient and permanent and exhibit as much as 1000x slowdown, and hence should be treated differently. Stragglers are mostly detected at the task level and mitigated by speculative execution [72, 187], which can suffer from several pitfalls (Section 5.3.1). Another tail-tolerant approach is cloning requests [21, 71]. If designed carelessly, cloned requests might involve the same limpware, exhibiting the same pitfalls as in Hadoop. Cloning is also limited to small jobs with little resource consumption.

Many solutions have been proposed to enforce performance isolation and fairness at various levels (*e.g.*, disk [171], CPU [188], VM [94], and cloud tenants [154]) and to manage performance variability using runtime adaptation techniques [26, 25, 95, 172]. In this thesis, we argue an end-to-end approach is needed; high-level performance management policies must incorporate individual low-level hardware performance.

Big data should flow in big pipes, but design flaws could introduce bottlenecks which lead to "small pipes" [102, 180], or in our case, limplock. To unearth design flaws, pinpoint implementation bugs, or diagnose misconfiguration, many approaches analyze system-specific information such as request flows [148], systems logs [181], and configuration snapshots [173]. We similarly use white-box metrics for manual diagnosis of limpware-intolerant designs. Other work leverages black-box metrics for statistical performance diagnosis (*e.g.*, CPU usage) [61, 64, 112]. Here, code debugging is a non-goal, and deep design flaws are hard to find.

Current cloud benchmarks (*e.g.*, YCSB [66], YCSB++ [139]) typically evaluate performance trade-offs among various cloud systems. Our work is complementary; limpbench evaluates the performance of cloud systems under limpware scenarios.

Chapter 8

Conclusion and Future Work

In this chapter, we first summarize the contributions of this dissertation (Section 8.1). We then list a set of lessons that we learned from years of researching the reliability of cloud systems (Section 8.2). Finally, we present future directions where our work could possibly be extended (Section 8.3).

8.1 Summary

Years of research has led to many failure-handling innovations, many of which are realized in today systems. For instance, RAID (Redundant Array of Independent Disks) has been used widely to tolerate disk failures [59, 141]. Logging and journaling appear in almost all systems, helping to tolerate machine crashes [90]. However, as we are moving into the cloud era, new challenges arise. The notion of *binary* hardware – hardware that either works correctly or stops functioning completely – no longer holds. Instead, hardware can suffer from corruption and performance degradation. Moreover, at a large scale, hardware fails more and more frequently.

As hardware failure gets more complex, cloud systems must be prepared. Unlike single-server systems, cloud systems are considerably more complex as they must deal with a wide range of distributed components, hardware failures, users, and deployment scenarios. It is not a surprise that they periodically experience downtime [119]. There is much room to improve the robustness of cloud systems.

This dissertation began with a simple question: why are cloud systems not 100% reliable? In our attempt to provide a clear answer, the question has led us to many more intricate questions. How can we verify the correctness of cloud systems in how they deal with the wide variety of possible failure modes? How can we build distributed systems that efficiently handle fail-silent failures? What are the impact of degraded hardware on today's systems? Finally, how should future generation systems handle this vexing failure mode effectively?

To address all the important questions above, this dissertation makes several contributions. First, with FATE and DESTINI, the recovery code of cloud systems is systematically tested in the face of multiple failures. Second, with SLEEVE, cloud systems are able to handle fail-silent behaviors efficiently. Third, with *limpware* analysis, we unearth many design flaws in current systems and show how they are unable to make limpware *fail in place*. Fourth, we present PMC, an effective approach to find limplock bugs in complex systems. Finally, we propose that an important area for future research is to build limpware-tolerant cloud systems.

We want to emphasize that these contributions should not be view as independent techniques. In fact, it is possible to use one technique with others, in order to maximize the potential and usefulness of each technique. For instance, to verify the correctness of limpware recovery, one could extend FATE and DESTINI

to support limpware injection. As another example, to build limplock detection and recovery service for existing systems, it is possible to extend the SLEEVE approach to incorporate *performance* model into the sleeved layer, enabling it to detect performance failures, in addition to fail-silent failures.

8.2 Lessons Learned

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

- **Expect the unexpected:** Large-scale cloud systems comprise a wide range of cooperating components that can fail *frequently*, in *different ways* (stop, corrupt, or *limp*), and in *any time* (*e.g.*, during a complex operation). Unfortunately, our experiences with open-source cloud systems reveal that the recovery path is often complex, under-specified, and tested less frequently than the normal path. Future systems must be designed for failure, and reliability has to be considered as a first class citizen when building new systems. The recovery path must be well tested, not just with offline (*i.e.*, before deployment) testing, but perhaps during operation as well [122]. Finally, testing framework must be able to cover different cases that may arise in real deployment.

- **Crash and reboot is occasionally expensive:** In distributed systems, crash and reboot is a traditional way to perform recovery of a faulty node. In fact, today's systems are built to crash, as user requests can be failed over to other healthy nodes and the system states can be reconstructed from persistent states and the healthy nodes across network. Although simple, this

strategy is sometimes not optimum. For example, in HDFS, we found that crashing the faulty NameNode and rebooting it may undesirably take several hours in a typical cluster. In MapReduce, crashing and rebooting a faulty TaskTracker may lead to large amount of tasks (*e.g.*, thousands) rerun. Perhaps, it is desirable for cloud systems to perform recovery in a more fine-grained manner (*e.g.*, rerun the faulty task, repair corrupted state, etc.) whenever it is possible. In this dissertation, we show the application of this micro-recovery principle to recover the faulty NameNode in few seconds (as opposed to hours).

- **The cure is sometimes worse than the disease:** Failure handling involves failure detection and recovery. However, if the root cause of a failure is wrongly detected, recovery action could exacerbate the situation. We encountered a case in HBase that when a region server dies because it incorrectly handles a corrupt region file. HBase fails over the task to another live region server, which unfortunately runs the *same* error handling code and will also die. The final outcome is that the entire cluster is down due to a single corrupt file. In MapReduce, a degraded network link connecting two racks causes exceptions when fetching intermediate data. MapReduce misinterprets the exception and assumes the machine storing the data is bad; it responses by blacklisting the machine and reruns all the tasks. The problem gets worse when all the tasks are relaunched on the same rack with the blacklisted node. Eventually, all the machines in the same rack are blacklisted. To reduce the severity of these problems, we believe that cloud systems must judiciously distinguish between different failure modes (*e.g.*, hardware failures vs. software bugs, stop vs. limp).

8.3 Future Work

The results we presented demand an era of limpware-tolerant cloud computing. This era requires a major evolution of systems principles, design, and implementation. One possible future work is to build an automated limplock detection and recovery service, which can be easily integrated with current cloud systems (Section 8.3.1). Another extension to our work could be a novel methodology to automatically build new cloud systems that are limpware-tolerant (Section 8.3.2).

8.3.1 Automated Limplock Detection and Recovery Service

To make current systems limpware-tolerant, one approach is to detect and recover from limplock automatically. Here, we present the possibility of building an *automated* limplock detection and recovery *service* (LDR). This LDR service must be able to detect limplock at various levels (operation, node, and cluster), pinpoint the culprit node, and perform proper recovery. For this LDR service to receive widespread adoption, gray-box techniques [23] could be explored to build the service. Gray-box LDR service does not require intrusive modifications of target systems. Instead, it can leverage knowledge about the systems (*e.g.*, data replication, LRU caching) to detect limplock scenarios. In the following subsections, we discuss potential challenges in building a gray-box LDR service.

Detection

LDR must be able to detect all levels of limplock, as accurate detection is fundamental for proper recovery. For each limplock level, LDR has to decide which metrics and statistics to use appropriately. Operation limplock often manifests

into prolonged execution time. Thus, LDR could detect operation limplock by monitoring operations' execution time. For instance, an operation experiences limplock when its execution time is greater than the 99th percentile.

Node limplock is more complicated to detect; it happens when all operations that must be served by the node experience limplock. LDR could track recent operations of a node over a *rolling time window*. If all operations in the time window experience limplock, then LDR could declare that the node experiences limplock. A challenge here is to decide how large the time window should be. If the time window is too large, the node may experience limplock for a long time before being detected and recovered. If the time window is too small, LDR may incorrectly detect a limplock situation (*i.e.*, a false positive) and trigger unnecessary recovery.

Another source of false positives is workload bursts, which could result in prolonged execution time of an operation. Various approaches to distinguish burst-induced vs. limpware-induced slowdown could be explored. For instance, in systems that use bounded thread pools for handling operations, LDR can leverage throughput metrics for this purpose. In burst-induced scenario, the throughput is often saturated at the maximum value that the system can provide in the normal case (*i.e.*, no limpware). On the other hand, in limpware-induced scenarios, the throughput often drops way below this maximum value.

Cluster limplock is possibly the most challenging to detect, as LDR must monitor all the nodes in the cluster in order to make decision. In general, cluster limplock happens when the master in a master-slave system experiences limplock (*e.g.*, ZooKeeper leader, HDFS namenode) or when all the nodes experience limplock (*e.g.*, HDFS cluster regeneration limplock in Section 5.3.2, Hadoop cluster limplock in Section 5.3.1). Detecting the former case is simpler as it can be narrowed

down to detecting master limplock. Detecting the latter case is trickier. In this case, LDR has to maintain a *global* snapshot of the cluster in order to make the call. Here, false positive can happen because of stale information; by the time that LRD analyzes the *global* snapshot to detect cluster limplock, the snapshot may not reflect the *true* state of the cluster at that time.

LDR must be able to correctly pinpoint the culprit node. This is fundamental for proper recovery. We believe that LDR must support *fine-grained* detection for this purpose. However, there are two major challenges. First, an operation could involve more than one node. For instance, consider an HDFS write pipeline containing three nodes, one of which is slow. If LDR only performs limplock detection at the HDFS client library level, it can only detect that a write is in limplock, but it could not pinpoint which node is slow. Second, layering (*e.g.*, Hadoop and HBase on top of HDFS) makes pinpointing the culprit node even more challenging. For example, a limplocked put operation in HBase could result from either a limping region server or limping HDFS datanode storing the transaction log of that server. Here, we believe a *cross-layer* detection is needed. Various approaches for LDR service to pinpoint the culprit node correctly and efficiently should be explored.

Recovery

Once the culprit node is pinpointed, LDR can perform appropriate recovery. As our results show that crashing is a good option in many cases, LDR can simply crash the culprit node. Here, LDR transforms limping failure into fail-stop failure, and leverages existing mechanisms of the target systems for recovery.

A major challenge is to judiciously trigger recovery once limplock is detected. In particular, LDR must decide the level of SLA violation at which it starts to per-

form recovery (*e.g.*, crashing). To do so, LDR must be able to estimate "recovery costs". If recovery is expensive (*e.g.*, the culprit node is a single point of failure and reboot may take a long time), it may be better to "limp" along than to crash. On the other hand, if recovery is cheap (*e.g.*, the target system has redundant resources to fail over), crashing is appropriate.

Various ways to estimate recovery costs should be investigated. One potential solution is to leverage gray-box knowledge about the target systems. For instance, if data is replicated, then crashing the limping node may not affect the SLA that much, as the system can quickly fail over to healthy nodes. As another example, if the culprit node is stateless (*e.g.*, task nodes in Hadoop), crashing the node is cheap (*e.g.*, Hadoop can re-execute the tasks elsewhere). However, if the culprit node is stateful, crashing the node may result in long recovery process (*e.g.*, the target system has to reconstruct the state by replaying the transaction log) and incur additional load (*e.g.*, regeneration of lost data).

8.3.2 Limpware-tolerant Cloud Systems

We believe systems designs employed by current cloud systems are inappropriate for handling limpware. First, these designs are originally created for high performance Internet services (*e.g.*, Web servers), thus they may no longer be suitable in today's cloud systems. Specifically, these systems employ either thread-per-request design, bounded thread pools, or staged event-driven architecture (SEDA). Our findings show that all of these designs (except SEDA) are inappropriate to tolerate limpware (Section 5.3.2, Section 5.3.5). For instance, they exhibit what we call the "false slowdown problem" (*e.g.*, a read from cache is heavily affected by a limping disk).

Second, these designs require administrators to manually set configuration parameters (*e.g.*, number of threads/queues), parameters that cannot be changed easily during runtime, making it hard to handle sudden changes such as workload bursts or the presence of limpware. Dynamic resource management mechanisms (*e.g.*, increasing the number of threads automatically), if any, are primary for overload management, hence inappropriate for dealing with limpware. Imagine a limpware-induced slowdown incorrectly detected as overload-induced. Here, recovery might react (incorrectly) by throttling the workload or increasing the number of threads as opposed to isolating the limpware.

We argue that today's cloud systems are much more complex and thus, they require new systems designs and resource management strategies. In particular, they perform diverse operations, from foreground workload (*e.g.*, read and write) to background workload (*e.g.*, compaction, memstore flushing, and regeneration). These operations often require different resources (memory, disk, network, and CPU), and can vary vastly in execution time. For instance, in HBase, a read from cache only takes a few microseconds, while a region flush could take tens of seconds; unfortunately, these two operations are still served by the same thread pool. Moreover, as these operations are typically *interdependent*, the impact of dependency can easily manifest during limpware scenarios. For instance, a slow background compaction could affect foreground read performance; a slow background memstore flushing, which foreground writes depend on, could make the foreground writes block waiting for available memory. Thus, if not carefully managed, a limplocked operation could easily affect others and cascade to node and cluster limplock.

These observations show that the current ways of building cloud systems are

broken under limpware scenarios. Therefore, one could potentially develop a novel framework that helps building limpware-tolerant cloud systems *automatically*. The high level idea is that the developers only need to *specify* the system they want to build. The framework then *transforms* this specification into runnable code that uses novel *design patterns* to isolate limplock and prevent limplock from cascading. We now elaborate the new methodology in detail.

- **System specification.** In this new framework, the developers have to specify *operations* that the system performs and their *dependencies*. An operation's dependencies can be either network calls to other nodes in the system, I/Os to persistent storage, or other operations. As these dependencies are potential sources of limping failure, they must be managed carefully. Various approaches such as declarative programming languages [126] could be used to specify systems.

- **Transformation.** Once the system is fully specified, the framework will transform the system's specification given by developers into runnable code automatically. The code will employ limplock isolation and avoidance design patterns. For instance, it could leverage differentiated thread pools to prevent limplock from cascading. Different operations will be served by different thread pools. With this design, a limping dependency can result in an exhausted thread pool, but it will not cascade to the whole system.

- **Limplock detection and recovery.** By design, cloud systems built with this new methodology are able to isolate limplock. Another benefit of building new systems from scratch is that limplock detection and recovery principles (Section 8.3.1) can be integrated from the very beginning. As a result, in

addition to isolating limplock, new cloud systems are able to detect and recover from limplock effectively.

- **Building new systems.** This new methodology could be applied to build new limpware-tolerant systems such as a distributed file system, and a key-value store. A distributed file system is a possible target as it is a substrate on which many scale-out cloud applications run (similar to HBase and Hadoop on HDFS). Building a key-value store running on top of the distributed file system will help demonstrate the concept of layering limpware-tolerant systems.

8.4 Closing Words

This dissertation is triggered by an important question of how cloud systems should effectively handle new, destructive, and diverse failure modes that are becoming more and more common. Decades of research portray how new failure modes always dramatically transform systems design and implementation. Likewise, we hope this dissertation helps bring new insights into rethinking reliability in the cloud and advancing the state-of-the-art of building reliable systems.

Appendix A

Impact of Limpware on HDFS: A Probabilistic Estimation

In this appendix, we calculate how often limplock scenarios happen in HDFS [9]. Although HDFS employs redundancies for fault-tolerance, its protocols are susceptible to limpware [77]. We specifically look at three protocols (*i.e.*, read, write, and regeneration) and quantify the probability that these protocols experience degraded condition. We further verify our calculation by simulation. Our results show that probabilities of these scenarios are alarmingly high in small and medium (*e.g.*, 30-node) clusters. However, these probabilities reduce significantly when size of cluster increases, as "Scale can be your friend" [137].

This appendix is structured as follows. We highlight an overview of HDFS in Section A.1, present the probability derivation in Section A.2, and conclude.

A.1 HDFS Overview

We now briefly describe the architecture and main operations of HDFS [9]. HDFS has a dedicated master, the *namenode*, and multiple workers called *datanodes*. The namenode is responsible for file-system metadata operations, which are handled by a fixed-size thread pool with 10 handlers by default. The namenode stores all metadata, including namespace structure and block locations, in memory for fast operations.

While the namenode serves metadata operations, the datanodes serve read and write requests. For fault tolerance, data blocks are replicated across datanodes. A new data block is written through a pipeline of three different nodes by default. Therefore, each data block typically has three identical replicas. On read, HDFS tries to serve the request a replica that is closest to the reader.

Since a data block can be under-replicated due to many reasons such as disk and machine failures, the namenode ensures that each block has the intended number of replicas by sending commands to datanodes, asking them to regenerate certain blocks. Block regeneration also happens when a datanode is decommissioned; all of its blocks are regenerated before it leaves. Each datanode allows at most two threads serving regeneration requests at a time so that the regeneration does not affect foreground workload.

A.2 Probability Derivation

In this section, we first show examples of limpware causing negative impact on three protocols of HDFS: read, write, and regeneration. We then calculate how often such scenarios happen for each protocol.

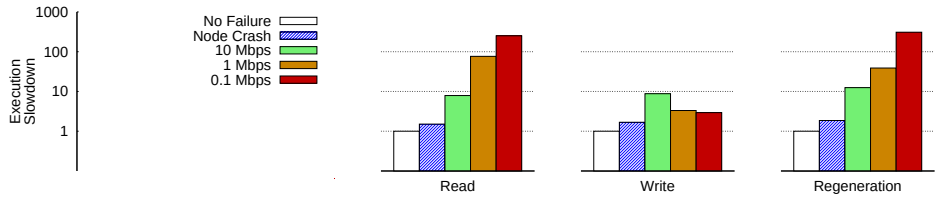

Figure A.1: **Impact of limpware on HDFS.** *The figures show the impact of a slow network card on three HDFS protocols: read, write, and regeneration.*

A.2.1 Impact of Limpware

Our previous work [77, 79] shows that HDFS is limpware intolerant. Here, we show examples of HDFS protocols suffering from negative impact of a slow network card (NIC). Specifically, we run workloads that exercise three HDFS protocols (read, write, and regeneration), inject slowdown to the NIC of a node in the cluster, and measure the resulting execution time. Figure A.1 shows the results. The normal bandwidth for the NIC is 100Mbps. We slow down the NIC to 10, 1, and 0.1Mbps in each experiment. We inject crashes to evaluate HDFS fail-stop failure tolerance. In all experiments, HDFS is not able to detect a slow NIC, hence does not trigger a failover. As a result, the total execution time in case of a slow NIC is orders of magnitude higher than in the normal scenario.

These results confirm that HDFS protocols are not able to tolerate limpware. We next quantify how often such negative impacts happen for each protocol, given the cluster's size, the number of data blocks it manages, and the number of user requests.

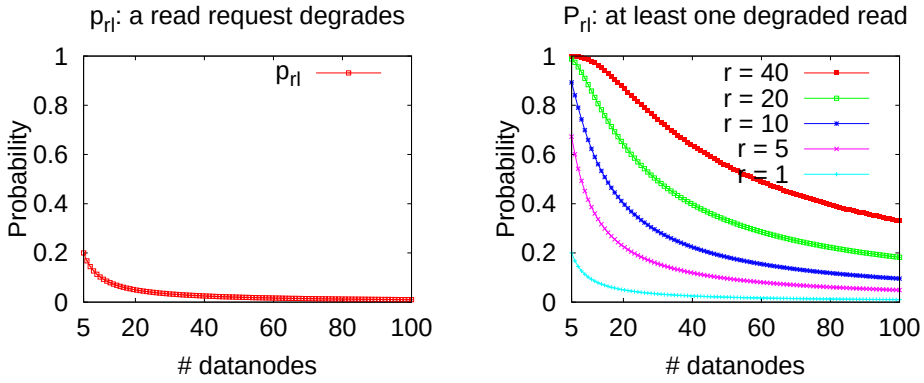

Figure A.2: **Degraded Read Probability.** *The figures show the probabilities for a request to degrade (p_{rl}) and for a user to experience at least one degraded read (P_{rl}).*

A.2.2 Degraded Read

• **Definition.** Consider an n-node cluster which has one slow node L and $n-1$ good nodes. A user request reads data from one out of three copies (assuming 3-way replication) of certain block B. Each copy has an equal chance to be chosen. We define a *degraded read* to be a read request that reads data the slow node L.

• **Derivation.** We now derive the probability of a degraded read. There are two conditions for a read of block B to degrade. First, L must contain one copy of B, and second, the copy in L is chosen for reading.

Let us derive the probability for the first condition. There are $\binom{n}{3}$ ways to choose 3 out of n nodes; there are $\binom{n-1}{3}$ ways to choose 3 out of $n-1$ *good* nodes. Therefore, the number of ways to choose 3 nodes, one of which is L, out of n nodes is $\binom{n}{3} - \binom{n-1}{3}$. The probability for L to contain one copy of B is:

$$P(L\ contains\ one\ copy\ of\ B) = \frac{\binom{n}{3} - \binom{n-1}{3}}{\binom{n}{3}} = \frac{3}{n} \qquad \text{(A.1)}$$

Since there are three copies of B, the probability for the copy in L to be chosen for reading is $\frac{1}{3}$. As a result, the probability for a read to degrade is:

$$P(a\ read\ to\ degrade) = p_{rl} = \frac{3}{n} \times \frac{1}{3} = \frac{1}{n} \tag{A.2}$$

Let r be the number of read requests of a user during a certain operation period (*e.g.*, a day). We now derive the probability that the user has at least one degraded read. The probability for a read *not* to degrade is $1 - p_{rl}$. The probability for *all* r requests *not* to degrade is $(1 - p_{rl})^r$. As a result, the probability for a user to experience at least one degraded read is:

$$P(user\ has\ at\ least\ one\ degraded\ read) = P_{rl} = 1 - (1 - p_{rl})^r = 1 - (1 - \frac{1}{n})^r \tag{A.3}$$

• **Result.** Figure A.2 plots the probabilities for a request to degrade (p_{rl}) and for a user to experience at least one degraded read (P_{rl}). As the cluster size increases, these probabilities decrease since there are more healthy nodes.

A.2.3 Degraded Write

• **Definition.** Consider an n-node cluster which has one slow node L and $n - 1$ good nodes. A write request requires HDFS to allocate 3 nodes to write to (assuming 3-way replication). Each node has an equal chance to be chosen in a write pipeline. We define a *degraded write* to be a write request whose pipeline contains L.

• **Derivation.** We now derive the probability for write to be slow. It is the probability for L to be chosen as one of the nodes in the 3-node write pipeline. We

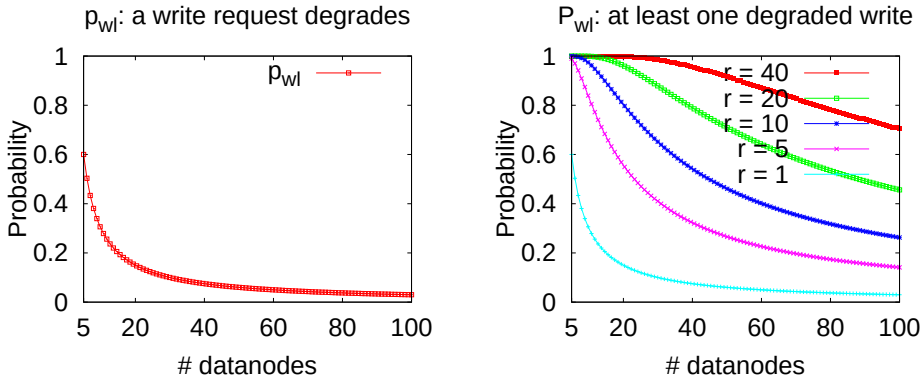

Figure A.3: **Degraded Write Probability.** *The figures plot the probabilities for degraded write as function of cluster size and the number of user requests.*

follow the similar derivation as in Section A.2.2. There are $\binom{n}{3}$ ways to choose 3 out of n nodes; there are $\binom{n-1}{3}$ ways to choose 3 out of $n-1$ *good* nodes. Therefore, the number of ways to choose 3 nodes, one of which is L, out of n nodes is $\binom{n}{3} - \binom{n-1}{3}$. Thus, the probability for a write to be degraded is:

$$P(a\ write\ to\ degrade) = p_{wl} = \frac{\binom{n}{3} - \binom{n-1}{3}}{\binom{n}{3}} = \frac{3}{n} \tag{A.4}$$

Let r be the total number of requests that a user has during a certain working period (e.g., a day). We now derive the formula for the probability that the user experience at least one slow write, P_{wl}. The probability for a write *not* to be slow is $1 - p_{rl}$. The probability for the user does *not* have any slow write equals the probability that *all* r write requests are *not* degraded, which is $(1 - p_{wl})^r$. Therefore, the probability for the user experiences at least one degraded write is:

$$P(user\ has\ at\ least\ one\ degraded\ write) = P_{wl} = 1-(1-p_{wl})^r = 1-(1-\frac{3}{n})^r \tag{A.5}$$

• **Result.** Figure A.3 plots the probabilities for degraded write as function of cluster size and the number of user requests. These probabilities are significantly larger than those for degraded read because each write has to be written to a 3-node pipeline. Even in a cluster of 50 nodes, a user is likely to experience one slow write every 40 requests.

A.2.4 Degraded Regeneration

Definitions

Consider an HDFS cluster consisting of n datanodes, one of which is slow (node L). Let C be a node that crashes; there are $n-1$ surviving nodes including the slow one. Let G be the set of good nodes (nodes are neither slow nor crashed); there are $n-2$ good nodes.

Let b be the total number of blocks in node C. When node C crashes, HDFS triggers a regeneration workload to regenerate those lost blocks. On average, each surviving node has to replicate $m = \frac{b}{n-1}$ blocks to other live nodes.

$$m = \frac{b}{n-1} \tag{A.6}$$

For each lost block, the master chooses a source and a destination datanode. The source is chosen from live nodes that still carry the block. The destination node is chosen using the write allocation policy (that uses randomness). A source datanode can only run two regeneration threads at a time.

• **Degraded node.** Consider a good node $X, X \in G$. When both regeneration threads of X send blocks to a slow node L, the node is not available for new regeneration tasks until the two threads finish (which could take a long time). We

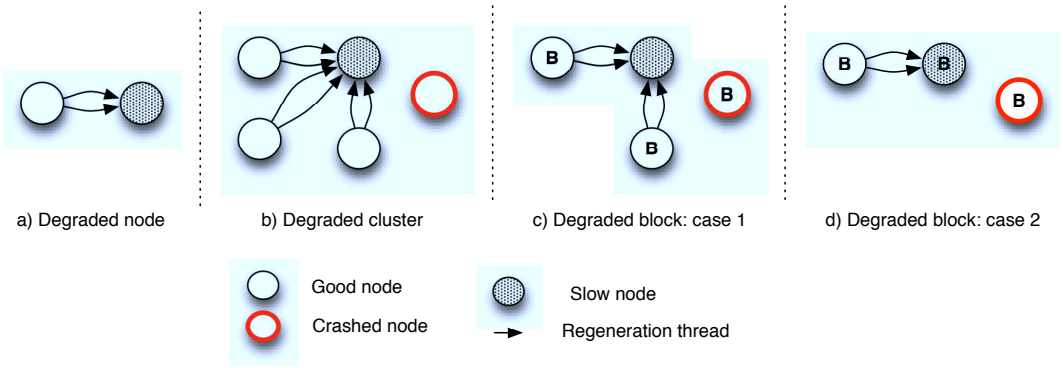

Figure A.4: **Degraded Regeneration.** *The figures show different scenarios of degraded regeneration. "B" label inside a circle represents that a copy of block B is located in the node. Arrow represents a regeneration thread, which may be copying a block other than B.*

define this situation a *degraded node* during regeneration process. The situation is illustrated in Figure A.4a.

• **Degraded cluster.** When *all* good nodes are degraded, as illustrated in Figure A.4b, the whole system is unable to start *any* regeneration task. We define this scenario a *degraded cluster*. Formally, the cluster degrades during regeneration when $\forall X \in G, X$ degrades.

• **Degraded block.** The system may not be able to regenerate block B for a long time. This can happen in two cases, which are illustrated in Figures A.4c and A.4d. First, all remaining copies of B are in degraded nodes (Figures A.4c), and second, one copy of B is in a degraded node, the other is in slow node L (Figures A.4d). Note that these cases are mutually exclusive and in the illustration, replication threads are copying different blocks other than B.

Derivation

We now derive the probabilities for degraded node, cluster, and block scenarios. To facilitate our calculation, we first derive the probability that node L is the destination of a copy task.

• L **is the destination for a copy task.** Consider a scenario where good node X ($X \in G$) copies one of its blocks (*e.g.*, block B) to another node. Let p be the probability that L is selected as destination. For this to happen, there are two conditions: first, L does not have a copy of B *and* second, the master chooses L to be the destination.

We now derive the probability of the first condition. Since X and C both contain a copy of B, the probability for L to also contain B is $\frac{1}{n-2}$. Therefore, the probability for a copy of B not in L is $1 - \frac{1}{n-2} = \frac{n-3}{n-2}$.

$$P(copy\ of\ block\ B\ in\ L) = \frac{1}{n-2} \tag{A.7}$$

$$P(copy\ of\ block\ B\ not\ in\ L) = 1 - \frac{1}{n-2} = \frac{n-3}{n-2} \tag{A.8}$$

We calculate the probability that the master chooses L as destination, given that L does not contain B. Note that the master can only choose one from $n-3$ nodes that do not have a copy of block B. Thus, given L not storing B, the probability for L to be the destination is $\frac{1}{n-3}$. As a result, the probability for X to copy block B to L is:

$$p = P(L\ is\ destination\ of\ a\ copy\ task) = \frac{n-3}{n-2} \times \frac{1}{n-3} = \frac{1}{n-2} \tag{A.9}$$

• **Degraded node probability.** Let P_{nl} be the probability for node $X, X \in G$, to degrade during regeneration process. We assume the time to copy a block between two good nodes is inconsiderable compared with the time to copy a block between a good node X and a slow node L. As a result, P_{nl} is the probability that X copies *at least* two blocks to L, out of m blocks it has to regenerate.

Since the probability for X *not* to copy *any* blocks to L (out of m blocks) is $(1-p)^m$ and the probability for X to copy *exactly* one block to L (again, out of m blocks) is $\binom{m}{1} \times p \times (1-p)^{m-1}$, we have:

$$\begin{aligned} P(a\ node\ degrades) &= P_{nl} \\ &= 1-(1-p)^m - \binom{m}{1} \times p \times (1-p)^{m-1} \\ &= 1-(1-\frac{1}{n-2})^{\frac{b}{n-1}} - \frac{b}{(n-1)\times(n-2)} \times (1-\frac{1}{n-2})^{\frac{b-n+1}{n-1}} \end{aligned} \tag{A.10}$$

• **Degraded cluster probability.** Let P_{cl} be the probability for the whole cluster to degrade during regeneration. This scenario happens when all good nodes degrade. Therefore:

$$P(the\ cluster\ degrades) = P_{cl} = {P_{nl}}^{n-2} \tag{A.11}$$

• **Degraded block probability.** Let p_{bl} be the probability for a block B to be degraded. There are two mutually exclusive cases for this scenario (Figure A.4c and Figure A.4d). Let the probabilities of these cases are p_{bl_1} and p_{bl_2}, respectively. Because they are mutually exclusive, we have:

$$p_{bl} = p_{bl_1} + p_{bl_2} \tag{A.12}$$

We now calculate probability of the first case, p_{bl_1}, the case where all remaining copies of block B are stored in degraded nodes (Figure A.4c). Let i be the number of good but degraded nodes. The probability to have *exactly* i good but degraded nodes is:

$$P(having\ i\ degraded\ nodes) = \\ p_{nl}(i) = \binom{n-2}{i} \times {P_{nl}}^{i} \times (1 - P_{nl})^{n-2-i} \tag{A.13}$$

For this first case to happen, there are two conditions: (1) $i \geq 2$; and (2) two copies of block B are stored among those i nodes. There are $\binom{n-1}{2}$ ways to place two copies of B among $n-1$ nodes (excluding the crashed one which must contain B). There are $\binom{i}{2}$ ways to place two copies of B among i degraded nodes. Therefore, the probability for two copies of B be in two (out of i) degraded nodes is $\frac{\binom{i}{2}}{\binom{n-1}{2}}$. As a result:

$$p_{bl_1}(i) = p_{nl}(i) \times \frac{\binom{i}{2}}{\binom{n-1}{2}}, 2 \leq i \leq n-2 \tag{A.14}$$

To calculate the exact value of p_{bl_1}, we must consider all possible values of i. Because i can vary from 2 to $n-2$, the final equation for the probability of the first case (Figure A.4c) is:

$$p_{bl_1} = \sum_{i=2}^{n-2} p_{nl}(i) \times \frac{\binom{i}{2}}{\binom{n-1}{2}} \tag{A.15}$$

Now, let us calculate, p_{bl_2}, the probability for the second case (Figure A.4d), which happens when: (1) $i \geq 1$; and (2) one remaining copy of B is in L, and the

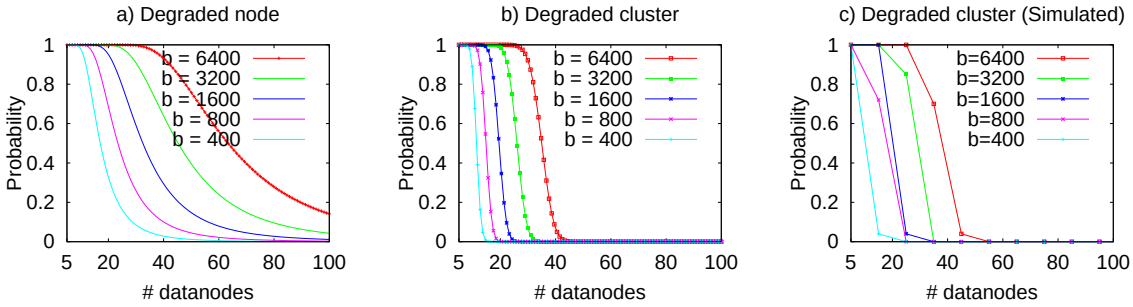

Figure A.5: **Degraded node and cluster probabilities.** *The figures plot the probabilities of a degraded node (a) and a degraded cluster (b). Figure (c) plots the simulation result for the probability of degraded cluster.*

other is in one (out of i) good but degraded nodes. Again, the probability to have *exactly* i good but degraded nodes is $p_{nl}(i)$. There are $\binom{i}{1} = i$ ways to place two copies of B, one of which in L and the other in a degraded node. Therefore, the probability for two copies of B be in this situation is $\frac{i}{\binom{n-1}{2}}$. As a result:

$$p_{bl_2}(i) = p_{nl}(i) \times \frac{i}{\binom{n-1}{2}}, 1 \leq i \leq n-2 \tag{A.16}$$

Since i can vary from 1 to $n-2$ in the second case, we have:

$$p_{bl_2} = \sum_{i=1}^{n-2} p_{nl}(i) \times \frac{i}{\binom{n-1}{2}} \tag{A.17}$$

Since two cases for a block to degrade are mutually exclusive, the degraded block probability is:

$$\begin{aligned} &P(a\ degraded\ block) = p_{bl} = p_{bl_1} + p_{bl_2} \\ &= \sum_{i=2}^{n-2} p_{nl}(i) \times \frac{\binom{i}{2}}{\binom{n-1}{2}} + \sum_{i=1}^{n-2} p_{nl}(i) \times \frac{i}{\binom{n-1}{2}} \end{aligned} \tag{A.18}$$

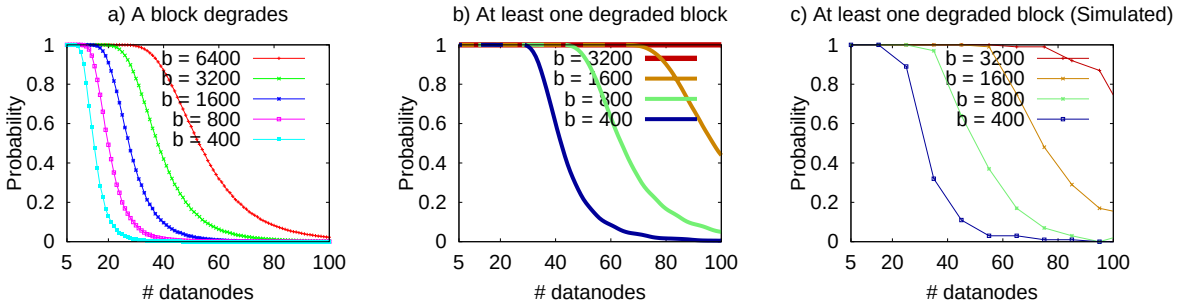

Figure A.6: **Degraded block probabilities.** *The figures show both the actual computation and simulation results for the probabilities of degraded block.*

We are now able to calculate the probability for the scenario where at least one block degrades during regeneration process. The probability for a block B not to degrade is $1 - p_{bl}$. The probability of having *zero* degraded block is $(1 - p_{bl})^b$. Therefore, the probability of having at least one degraded block is:

$$P(at\ least\ one\ degraded\ block) = P_{bl} = 1 - (1 - p_{bl})^b \tag{A.19}$$

Results

To be more confident with our calculation, we simulate the HDFS regeneration protocol and run a regeneration workload. We vary the number of nodes in the cluster and the number of lost blocks. We run each configuration (with different cluster size and number of lost blocks) 100 times and measure the probability of degraded block and degraded cluster. Figures A.5 and A.6 show both our calculation and the simulation results. The probabilities of degraded node and degraded cluster are relatively high for a small to medium (e.g., 30-node) cluster. The probability of degraded block is alarmingly high: even in a 100-node cluster, a dead 20%-full 1TB node (that can store 3200 blocks) will lead to at least one degraded block. Simulation results are similar to our calculation.

A.3 Conclusion

Limpware without doubt is a destructive failure mode, yet we show that HDFS fails to properly handle limpware. We present a probabilistic estimation of how often such negative impact of limpware happens to three important HDFS protocols: read, write, and regeneration. Our estimation shows that the impact of limpware is significant, even in a medium sized cluster of 30-40 nodes.

Appendix B

Hadoop Model In Detail

In this appendix, we present the Petri net model of the Hadoop speculative execution protocol. We present an overview of the model and discuss the JobTracker and TaskTracker modules in detail.

B.1 Overview

We now describe how we use Petri net to model Hadoop using a top-down approach. We first analyze the Hadoop design documentation [1] to understand the high-level Hadoop architecture in order to create main modules of the model. The Hadoop framework consists of a single master JobTracker and multiple slave TaskTrackers. The master is responsible for scheduling the tasks on the slaves, monitoring them, and responding to failures (*e.g.*, re-executing the failed tasks). The slaves execute the tasks as scheduled by the master and inform the masters about the task status.

Given this Hadoop high-level architecture, we construct two main modules

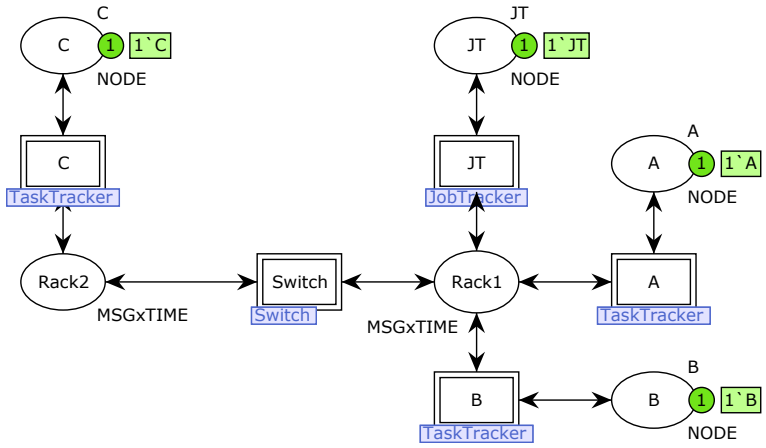

Figure B.1: **Model of a Hadoop Cluster.** *This figure is an example of a high-level model of a Hadoop cluster. It contains a single JobTracker and three different TaskTrackers in a cluster of two racks.*

(JobTracker and TaskTracker) for our Hadoop model. Figure B.1 is an example of the *top* page of a Hadoop model. It models a cluster with a single JobTracker and three different TaskTrackers. These nodes communicate with each other using messages passed through the network. Thus, each node is modeled with two network interfaces (denoted `PortIn` and `PortOut`), which model the processes of receiving and sending messages, together with a *core* module, which models the core logic of the node. This separation between the core logic and network interfaces enables us to perform limpware injection (*e.g.*, a slow NIC) easily. Figure B.2 represents the high-level model for the JobTracker node.

To obtain finer granularity for the main modules, we inspect the Hadoop source code to understand internal details of the JobTracker and TaskTracker. In particular, we inspect the data structures they maintain (*e.g.*, job, task, progress, and node information), the protocol messages they use to communicate with each other, the way that messages are handled, and the failure handling mechanisms (*e.g.*, timeout, retry, and speculative execution). With this understanding, we

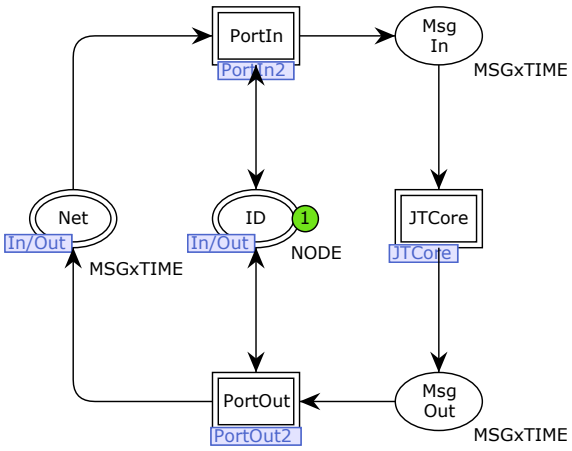

Figure B.2: **High-level Model of the JobTracker.** *The places* `PortIn` *and* `PortOut` *model the processes of receiving and sending messages. The place* `ID` *models the IP address of the JobTracker. Finally, the module* `JTCore` *model the JobTracker's* core *logic.*

leverage design patterns of model construction to gradually fill in the details for each module. In some situations, this process is straightforward. For instance, data structures representing information of a task such as type (map or reduce), data location, current state (*e.g.*, pending, running, failed, and complete), running location, and progress can be directly represented by complex color sets with multiple fields in Petri net. Figure B.3 represents the color sets that we use in our Hadoop model. We defer discussing these types until we present our model in details.

In other situations, we have to convert Hadoop source code into Petri net's ML functions to represent a particular computation (*e.g.*, compute a task to speculate). We could have built a Petri net module itself to represent such computation. However, doing so would blow up the complexity of the model and make it impractical for model checking later on. In the next subsequent sections, we describe in detail the Petri net models for the JobTracker and TaskTracker.

```
colset NODE = with JT | A | B | C | D | NULLNODE | R12 | R21;
colset NODExTIME = product NODE * TIME timed;
colset NODELIST = list NODE;
colset TASKTYPE = with M | R; (* Map/Reduce *)
colset JOB = product STRING * INT * INT;
colset JOBxINTxINT = product JOB * INT * INT;
colset NUM = INT;
colset ATTEMPT = INT;
colset PARAMS = NODELIST;
colset PROGRESS = INT;
colset TASKID = product JOB * TASKTYPE * NUM;
colset TASK = product JOB * TASKTYPE * NUM * ATTEMPT;
colset TASKxTASKID = product TASK * TASKID timed;
colset TASKxTASKIDxNODE = product TASK * TASKID * NODE;
colset TASKxTASKIDxNODExNODExTIME =
      product TASK * TASKID * NODE * NODE * TIME timed;
colset TASKIDLIST = list TASKID;
colset TASKxTASKIDLISTxNODExTIME =
      product TASK * TASKIDLIST * NODE * TIME timed;
colset TASKxPARAMS = product TASK * PARAMS;
colset TASKxPARAMSxINT = product TASKxPARAMS * INT;
colset NODExTASK = product NODE*TASK;
colset NODExTASKxPARAMSxPROGRESSxTIME =
      product NODE * TASK * PARAMS * PROGRESS * TIME;
colset RUNNINGTASK = list NODExTASKxPARAMSxPROGRESSxTIME;
colset TPLIST = list TASKxPARAMS;
colset STRINGxTPLIST = product STRING * TPLIST;
colset JOBxTPLIST = product JOB * TPLIST;
colset TASKxPARAMSxNODE = product TASK * PARAMS * NODE;
colset TASKxPARAMSxNODExTIME = product TASK * PARAMS * NODE * TIME timed;
colset TASKxPARAMSxPROGRESSxTIME = product TASK * PARAMS * PROGRESS * TIME;
colset TOA = TIME; (*done time*)
colset TDN = TIME; (*done time*)
colset TASKINFO = list TASKxPARAMSxPROGRESSxTIME;
colset DATA = INT;
colset TASKxDATAxTIME = product TASK * DATA * TIME timed;
colset TASKxTIME = product TASK * TIME;
colset UNITxTIME = product UNIT * TIME timed;
colset UNITxINT = product UNIT * INT timed;
colset JOBxTIME = product JOB * TIME;
```

Figure B.3: **Definitions of Color Sets.** *The figure shows the definitions of all data types used in the Hadoop model.*

B.2 JobTracker Model

Figure B.4 represents a Petri net model for the JobTracker. It is essentially an RPC server that communicates with clients (for job submission) and TaskTrackers (for job management and failure handling). For each RPC message, the JobTracker runs an RPC handler and returns the result. We investigate this RPC communication protocol in Hadoop source code and model it using the types `OPT` and `MSG` (defined in Figure B.5). Here, `OPT` is a *union* type that represents different operations between the nodes; `MSG` is a *record* type representing actual RPC messages floating around the network with four different fields that represent the sender, receiver, detailed operation, and size of a message.

In general, for each type of message, there will be a transition representing the corresponding action that handles the message. To keep our model small and checkable, we make two important simplifications. First, we model the parts that are relevant to job scheduling and speculative execution only. Second, we eliminate the interaction between clients and the Job Tracker. We describe each of these simplifications next.

The operations related to job scheduling are heartbeat processing, task scheduling, and speculative computation. A heartbeat from a TaskTracker contains status information about the sender such as the information of running tasks and the number of available task slots. The heartbeat handler is modeled using `Process HB` transition. When this transition occurs, the states of all the running tasks in the cluster (the place `Task State`) are updated; the heartbeating node is also put into a *wait-for-scheduling* state (the place `Wait`). Scheduling action is modeled using the transition `Schedule`. This transition takes a new task from the task queue (modeled as the place `New Tasks`) and assigns it back to the heartbeat-

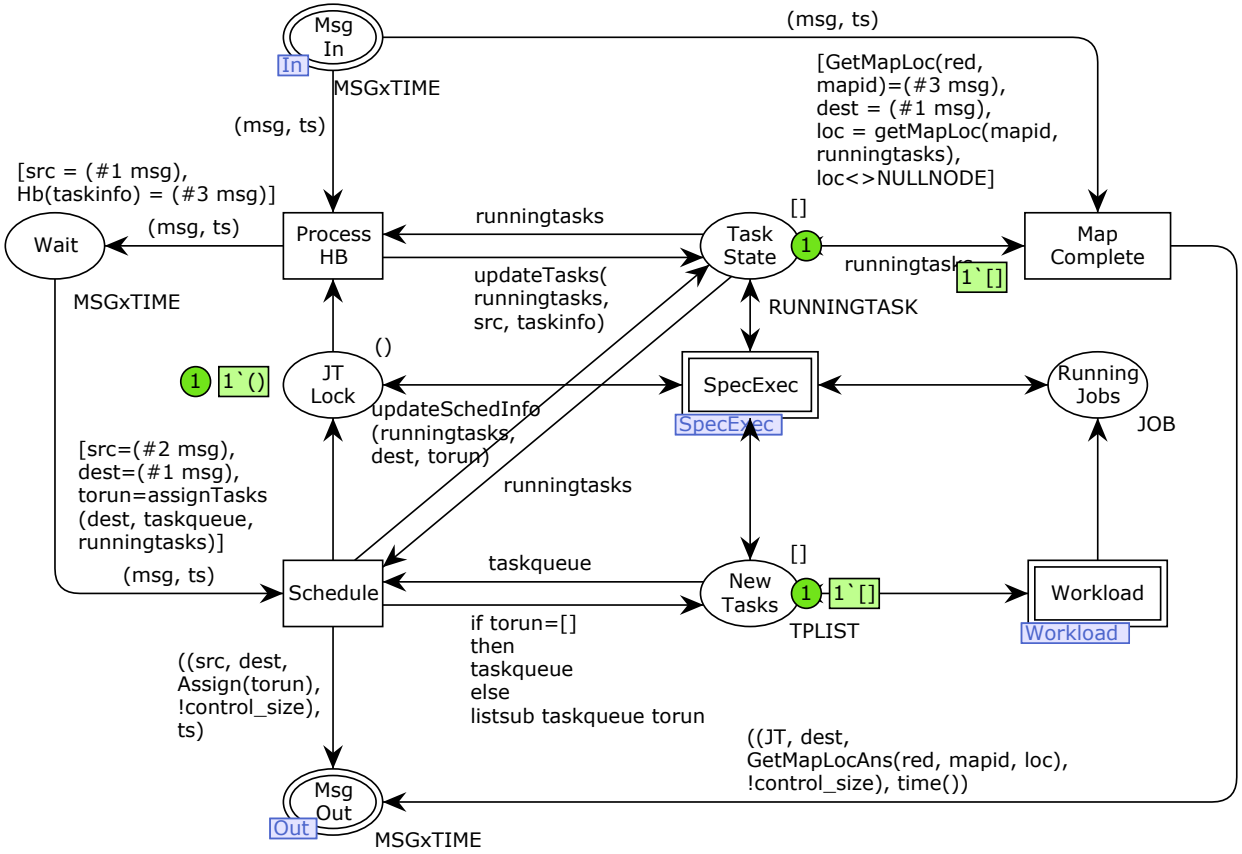

Figure B.4: **The JobTracker Core Model.** *This figure represents the Petri net that models the logic of the JobTracker in detail.*

```
(* Possible Operations *)
colset OPT = union
      Assign: TPLIST +
          Hb: TASKINFO +
    GetSplit: TASK + GetSplitAns: TASK +
       Write: TASK + WriteAns: TASK +
   GetMapLoc: TASKxTASKID + GetMapLocAns: TASKxTASKIDxNODE +
GetMapOutput: TASKxTASKID + GetMapOutputAns: TASKxTASKID;

(* RPC messages *)
colset MSG = product NODE * NODE * OPT * SIZE timed;
```

Figure B.5: **Modeling RPC messages.** *This figure shows the definition for RPC color set used in the Hadoop model.*

ing node (modeled as the out token to place `Msg Out`). When the transition `Schedule` occurs, it updates the place `Task State`. In addition to processing heartbeats, the JobTracker also processes "Get Map Complete" messages from reduce tasks asking for locations of immediate data. This is modeled using the transition `Map Complete`.

Astute readers may notice that we use quite a few ML functions in the model. One example is the function `assignTasks` (defined in Figure B.6) that appears in the *guard* of the transition `Schedule`. The function is deterministic; it basically takes three parameters (*i.e.*, the heartbeating node, the queue of new tasks, and the states of all running tasks in the cluster) and returns a task that the heartbeating node should run. What important here is the scheduling policy, which we model by interpreting the policy from Hadoop source code. That is, when choosing a task to schedule for a node, Hadoop favors map tasks (function `findMap`) over reduce tasks (function `findRed`) of the same job. Moreover, among the map tasks, Hadoop favors ones that are local to the node (*i.e.*, that is the input data for the tasks is located in the node), as modeled by function `findMapLocal`.

```
(* Find a local map task *)
fun findLocalMap(node, [], runningtasks) = []
  | findLocalMap(node,
    ((job, tp, num, attempt), splits)::taskqueue, runningtasks) =
  let
    val assigned = hasTaskInstance(node, (job, tp, num), runningtasks)
  in
    if tp = M andalso (mem splits node) andalso assigned=false
    then [((job, tp, num, attempt), splits)]
    else findLocalMap(node, taskqueue, runningtasks)
end;
(* Find a non-local map task *)
fun findMap(node, [], runningtasks) = []
    | findMap(node,
              ((job, tp, num, attempt), splits)::taskqueue, runningtasks) =
  let
    val assigned = hasTaskInstance(node, (job, tp, num), runningtasks)
  in
    if tp = M andalso assigned = false
    then [((job, tp, num, attempt), splits)]
    else findMap(node, taskqueue, runningtasks)
end;
(* Find a reduce task *)
fun findRed(node, [], runningtasks) = []
   | findRed(node,
             ((job, tp, num, attempt), params)::taskqueue, runningtasks) =
  let
    val assigned = hasTaskInstance(node, (job, tp, num), runningtasks)
  in
    if tp = R andalso assigned = false
  then [((job, tp, num, attempt), params)]
  else findRed(node, taskqueue, runningtasks)
end;
(* Find a task for a given node *)
fun assignTasks(node, taskqueue, runningtasks) =
  let
    val localMap = findLocalMap(node, taskqueue, runningtasks)
    val map = findMap(node, taskqueue, runningtasks)
    val red = findRed(node, taskqueue, runningtasks)
  in
  if localMap <> [] then localMap
  else if map <> [] then map
  else if red <> [] then red
  else []
end;
```

Figure B.6: **The Hadoop Scheduling Policy.** *This figure presents the ML functions that models the scheduling policy of Hadoop.*

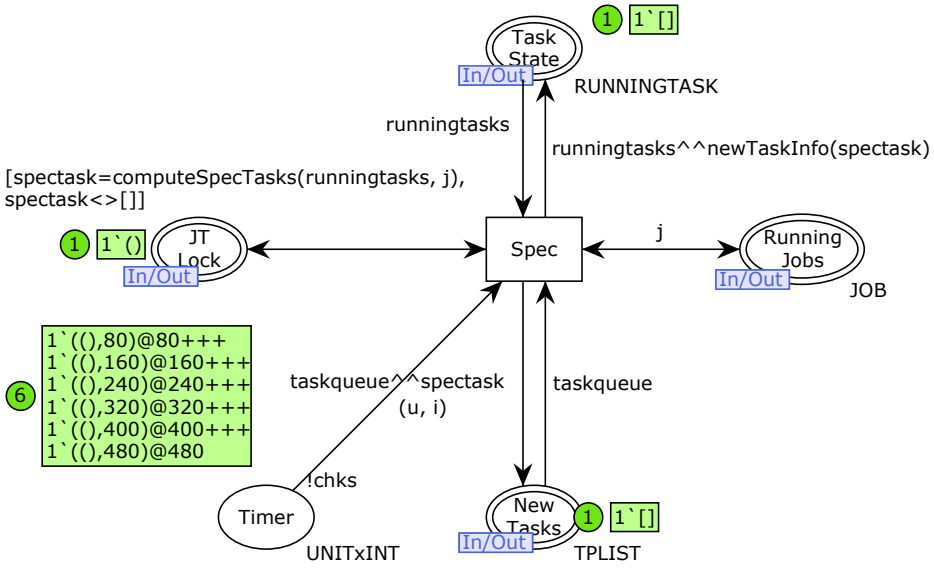

Figure B.7: **The Model of Speculative Execution.** *This figure presents the Petri net model of the* `SpecExec` *module.*

The ML functions closely follow this policy. This is one of the examples where modeling a system policy into Petri net language is not straightforward. Again, we could have created a Petri net model (with places and transitions) to just to emulate the computation. However, as we mentioned in Section 6.4, doing so would be challenging and impractical due to the state space explosion. As a result, when modeling this computation, we choose to translate the Java code of Hadoop to the ML code and embed it in our Petri net model.

Such translation is safe and correct (*i.e.*, we do not trade correctness for simplicity) for two reasons. First, as explained above, the computation is *deterministic*. Second, it is *atomic*, as the whole computation is protected by a global lock (modeled as the place `JT Lock`). Because of these two reasons, we do not lose accuracy when performing model checking later on. That is, we do not reduce the

```
(* Get all the task info of a job *)
fun getTasks([], job) = []
  | getTasks((node, (j, t, num, a), params,p, ts)::l, job) =
  if j = job
  then ((j, t, num,a), params, p, ts)::getTasks(l, job)
  else getTasks(l, job);
(* Get all the tasks that have already been schedule to run *)
fun getScheduledTasks([], job) = []
  | getScheduledTasks((node, (j, t, num, a), params,p, ts)::l, job) =
  if j = job andalso ts > intToTime(0) andalso node <> NULLNODE
  then ((j, t, num,a), params, p, ts)::getScheduledTasks(l, job)
  else getScheduledTasks(l, job);
fun avgProgress(tasklist, tpe) =
let
   fun size ([], tpe) = 0
       | size(((_, t, _, _),_,_,_)::tail, tpe) =
           if tpe = t then 1 + size(tail, tpe) else size(tail, tpe);
   fun sum ([], tpe) = 0
       | sum(((_, t, _, _),_,p,_)::tail, tpe) =
           if tpe = t then p + sum(tail, tpe) else sum(tail, tpe);
in
   if size(tasklist, tpe) = 0 then 0
   else sum(tasklist, tpe) div size(tasklist, tpe)
end;
(* Find a task to speculate, given average progress score *)
fun findSpecTasks([], tpe, avg, tasks) = []
   | findSpecTasks(((j, t, n, a), params, p, _)::tail, tpe, avg, tasks) =
  let val numattempts = numAttempts(tasks, (j, t, n)) in
    if tpe <> t orelse numattempts >= 2 then
         findSpecTasks(tail, tpe, avg, tasks)
    else if p >= avg then
         findSpecTasks(tail, tpe,avg, tasks)
    else ((j, tpe, n, a+1), params)::findSpecTasks(tail, tpe, avg, tasks)
end;
(* Compute a speculative task for a job *)
fun computeSpecTasks(runningtasks, job) =
  let
    val tasks = getTasks(runningtasks, job)
    val scheduledtasks = getScheduledTasks(runningtasks, job)
    val avgMap = avgProgress(scheduledtasks, M)
    val avgRed = avgProgress(scheduledtasks, R)
  in
    if allTasksRunning(scheduledtasks, job) = false then []
    else
      findSpecTasks(tasks, M, avgMap, tasks)^^
      findSpecTasks(tasks, R, avgRed, tasks)
end;
```

Figure B.8: **The Speculative Execution Policy.** *This figure presents the ML functions that models the speculative computation in Hadoop.*

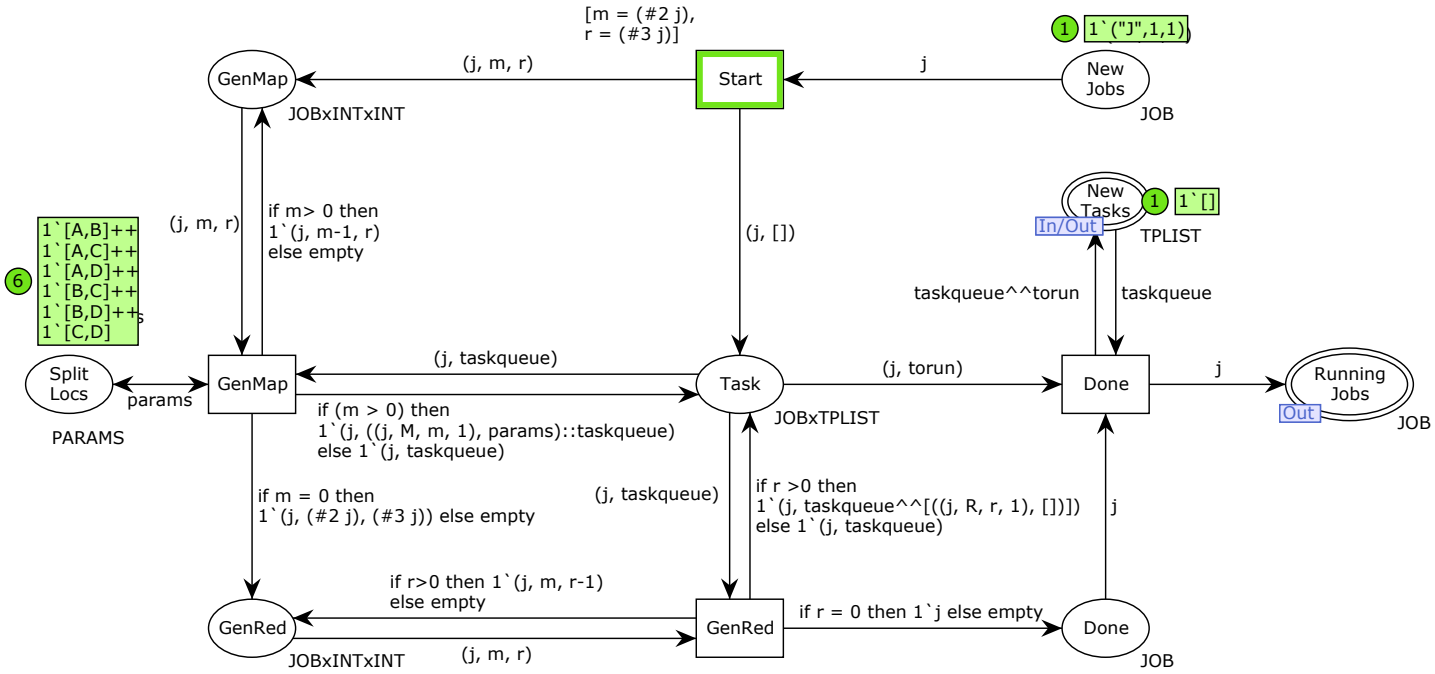

Figure B.9: **Workload Generator Model.** *The figure illustrates a Petri net model that generates tasks for JobTracker to schedule*

state space and miss some potentially interesting edge cases because we have not eliminated any nondeterminism.

The same argument applies when we model the speculative computation (illustrated in Figure B.7), a mechanism to handle slow tasks (*i.e.*, stragglers). In this model, the computation is modeled using the transition `Spec`. The place `Timer` is used to control how often the transition should occur. Here, we use a ML function `computeSpecTasks` to model the computation. Again, because this computation is deterministic and atomic, we simply translate the speculation policy that we infer from Hadoop source code into the ML functions. Specifically, Hadoop considers triggering speculative computation for jobs whose tasks all have been running. Moreover, it speculatively executes tasks whose current progresses are failing behind the average progress. Finally, it does not consider tasks that already have a backup task. All of this logic is encoded in the ML functions defined in Figure B.8.

In order to simplify the model of JobTracker, we not only focus on modeling the relevant parts of the speculative execution protocol, but also choose not to model the interaction with the clients. Instead, we create a `Workload` module that is responsible for generating new jobs. The `Workload` module is illustrated in Figure B.9. The place `New Jobs` with a color set `JOB` models various information of a job (*e.g.*, jobID and the number of tasks). We randomly generate the input data locations of map tasks (modeled as `Split Locs` place).

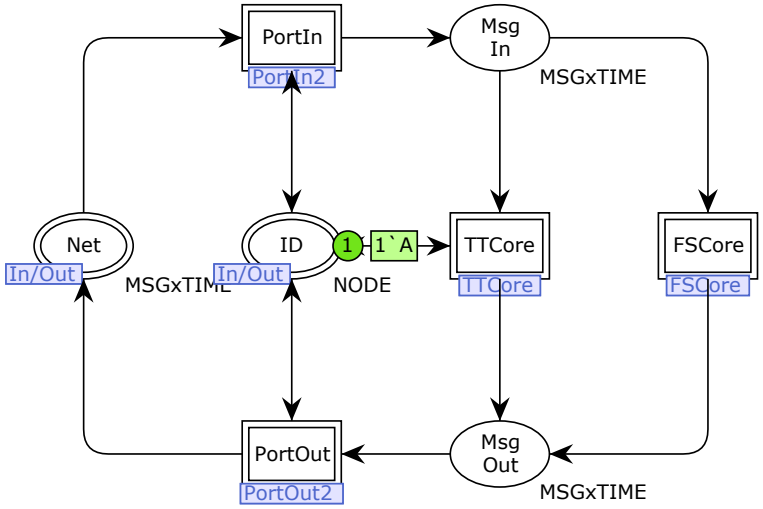

Figure B.10: **High-level Model of The TaskTracker.** *The places* `PortIn` *and* `PortOut` *model the processes of receiving and sending messages. The place* `ID` *models the IP address of the TaskTracker. Finally, the modules* `TTCore` *and* `FSCore` *model the task execution and HDFS read/write logic, respectively.*

B.3 TaskTracker Model

Figure B.10 illustrates the high-level model of a TaskTracker. Similar to that of the JobTracker, the model has the place `Net` for messages floating around the network, `ID` for the node's IP address, `Msg In` and `Msg Out` for input and output message buffers. It also has `PortIn` and `PortOut` modules that model the machine network card. The model has two core modules: `TTCore`, representing the TaskTracker logic (*e.g.*, run map and reduce tasks), and `FSCore`, representing the HDFS logic (*e.g.*, read and write data files). We choose to model HDFS logic within a TaskTracker model because in a typical deployment, a physical machine often runs these services at the same time.

The primary tasks of a TaskTracker are to receive instructions from the Job-

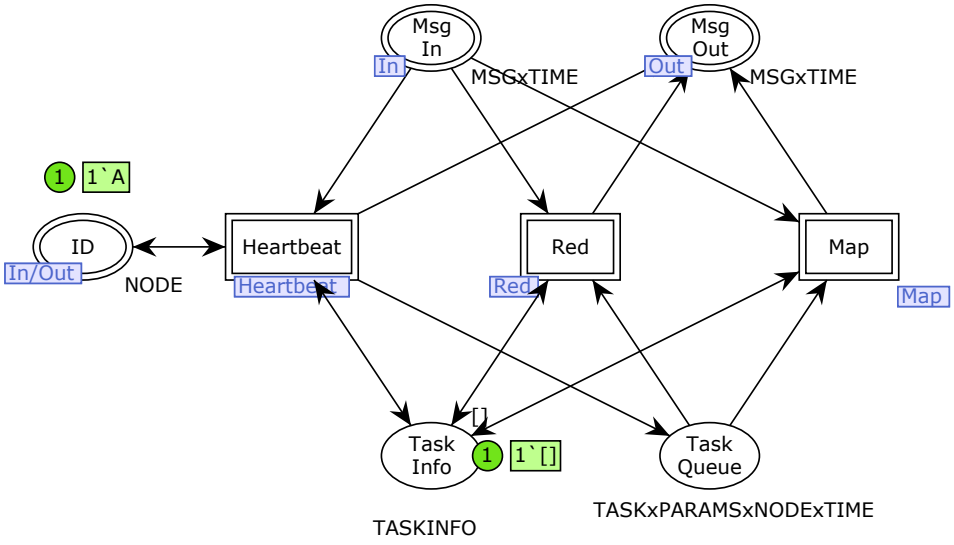

Figure B.11: **The TaskTracker Core Module.** *This figure represents the Petri net that models the logic of a TaskTracker. It contains main modules for sending heartbeats and executing map and reduce tasks.*

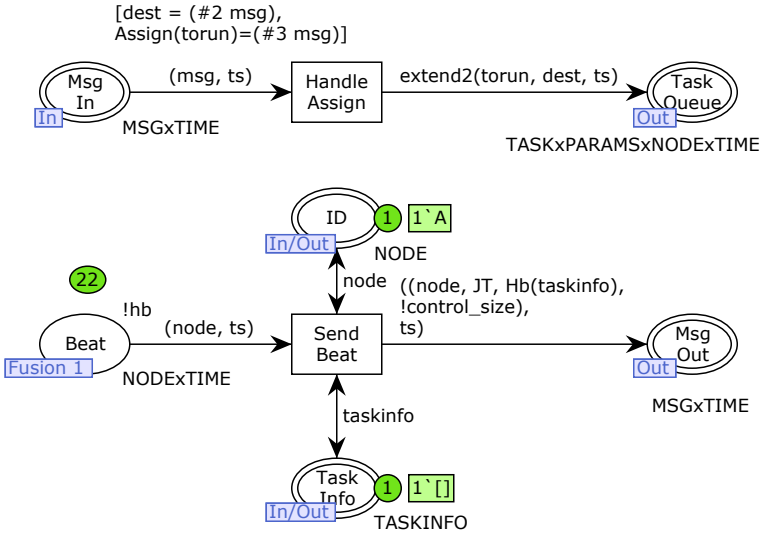

Figure B.12: **Heartbeat Model.** *This figure models the process of sending heartbeat of a TaskTracker.*

Tracker, execute tasks, and report back the task status. Its core logic is modeled in Figure B.11. It has two primary data structures: `Task Queue`, representing new tasks to be run, and `Task Info`, representing status of all the tasks that are currently running in this node. Communication with the JobTracker (to receive new tasks and report back the task status) is done via the `Heartbeat` module (Figure B.12). This module simply sends a heartbeat every fixed amount of time unit (controlled by the place `Beat` and the transition `Send Beat`). Upon a heartbeat ack (`Handle Assign` transition), it puts any new task to the task queue.

The logic to run map and reduce tasks is represented by two modules: `Map` and `Red`. Figure B.13 models the execution of a map task. It first checks if the input data for the task is local using the transition `Check local`. If the input data is not local, it then chooses a remote location to read from nondeterministically (transition `Select Loc`). It then sends a read request to this remote location to get the data (the transition `Read`). Finally, it processes the data (not shown).

Figure B.14 models the shuffling phase of a reduce task. It first finds the locations of finished map tasks by contacting the JobTracker (the transition `Lookup`). It then reads the map output data locally if the map task happens to be executed in the same machine (transition `Fetch Local`) or remotely (transition `Fetch Remote`). Here, we model the lookup and read retry logic to anticipate the scenario where the reduce tasks try to look up and read from an unfinished map task. Merge and reduce phases are modeled in Figure B.15 by transitions `Merge` and `Reduce` respectively. The locations of output files are picked randomly using the place `Dest`. Task progress is updated after every task action; this update is modeled as arcs coming in (for read) and out (for modification) of the place `Task Info`.

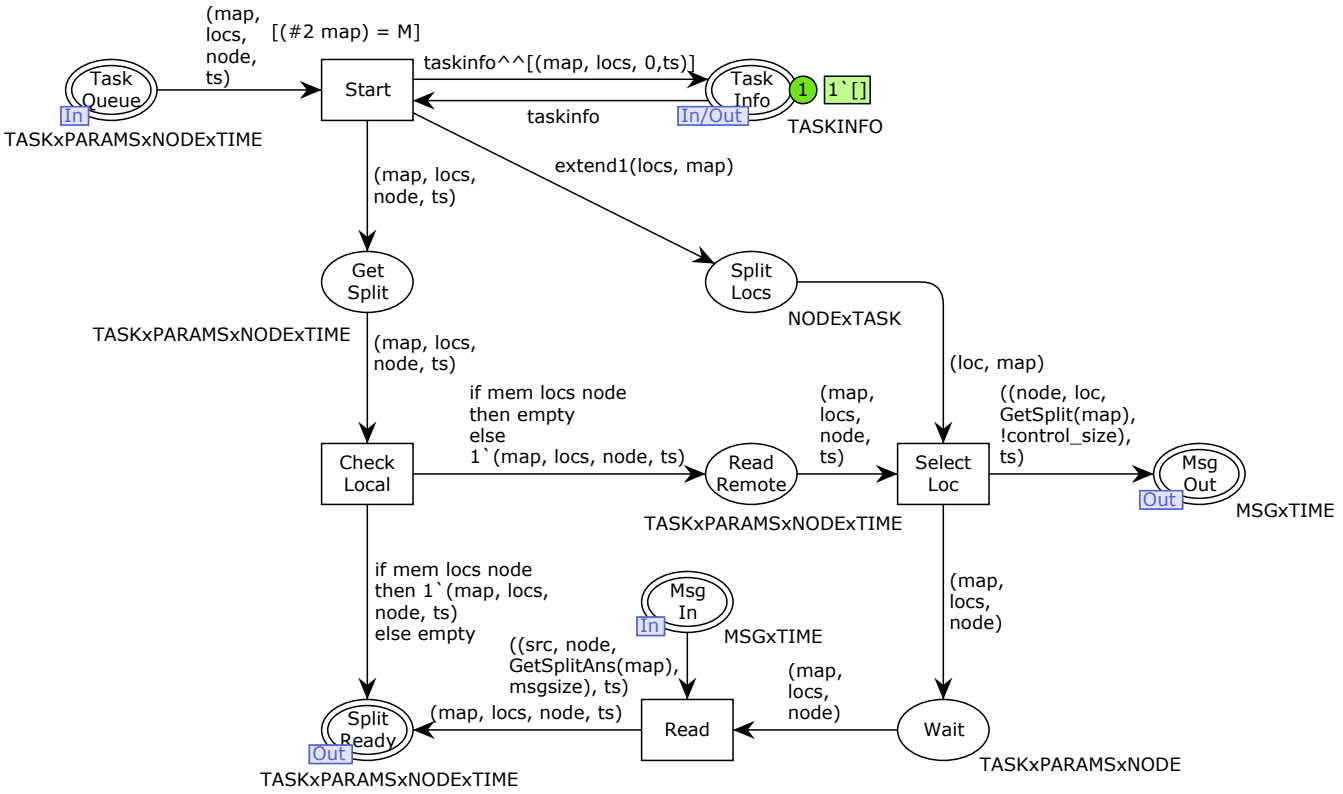

Figure B.13: **The Map Model.** *This figure presents the Petri net model of the reading data process of a map task.*

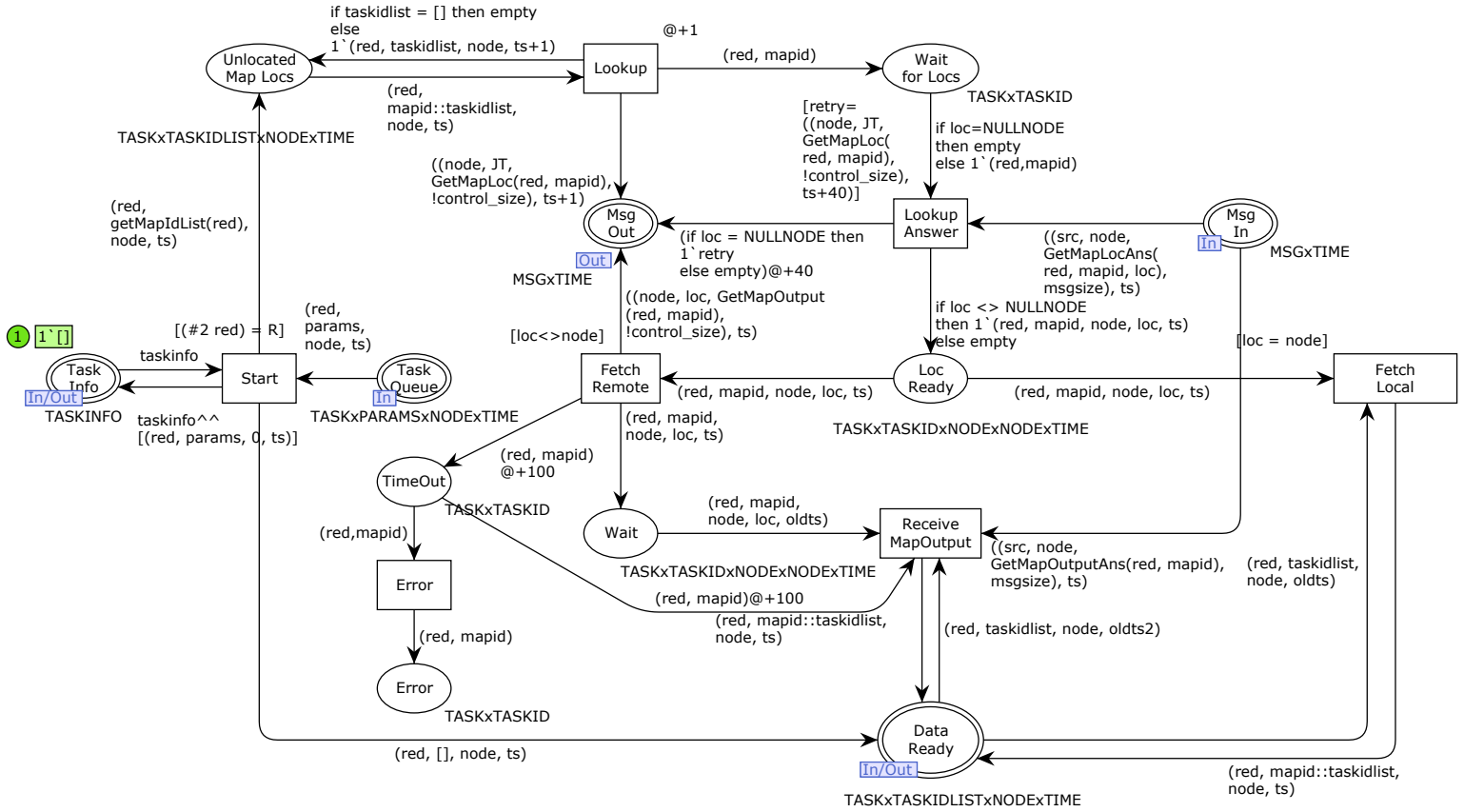

Figure B.14: **The Shuffling Phase Model.** *This figure presents the Petri net model for the shuffling phase (copying output data from of map tasks) of a reduce task.*

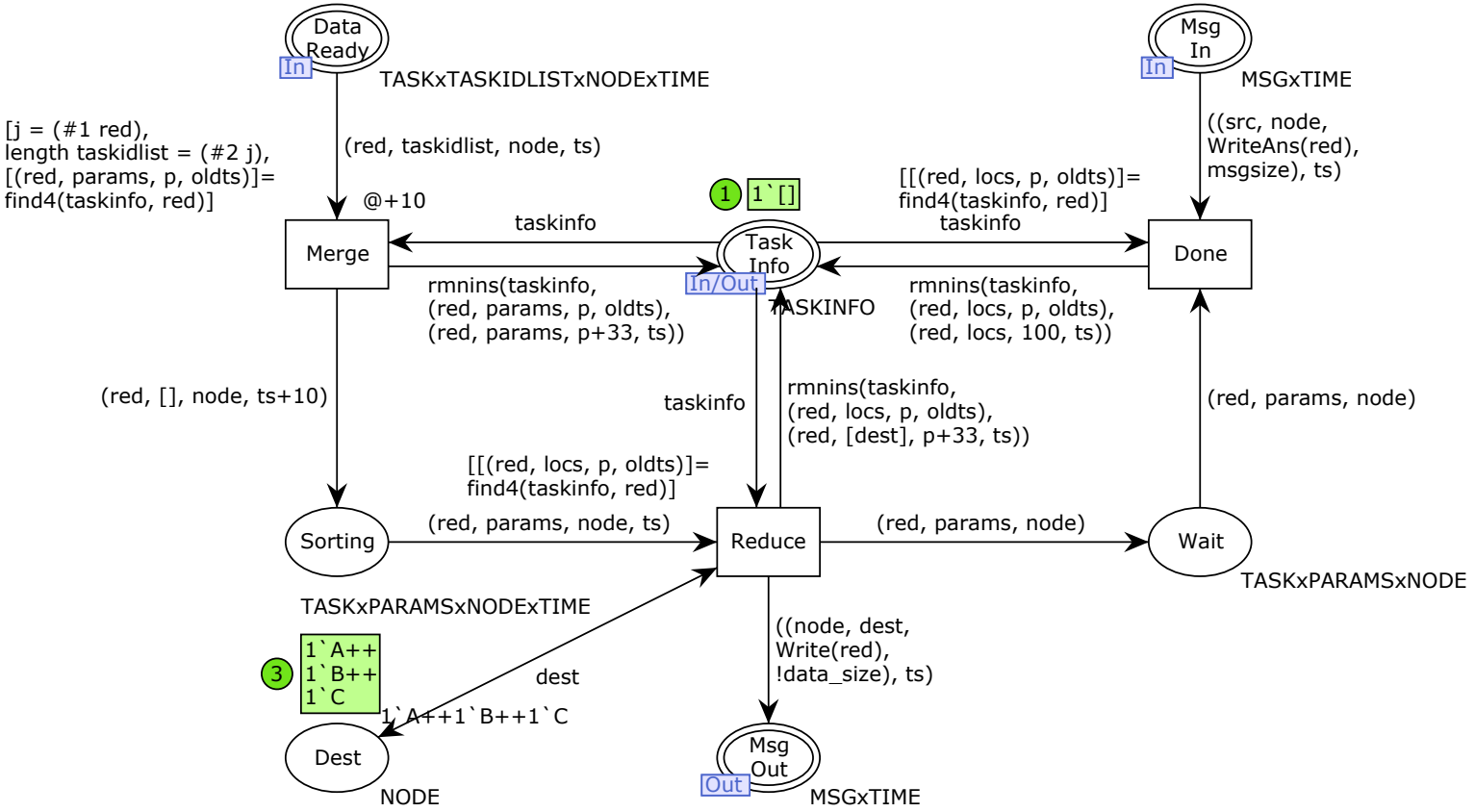

Figure B.15: **The Reduce Phase Model.** *This figure presents the Petri net model for the actual reduce phase (after data from map tasks are ready).*

Bibliography

[1] Apache Hadoop Documentation. http://hadoop.apache.org/docs/current/.

[2] Apache Hadoop Project Web Page. http://hadoop.apache.org.

[3] Apache HBase Project Mailing Lists. http://hbase.apache.org/mail-lists.html.

[4] Apache HBase Project Web Page. http://hadoop.apache.org/hbase.

[5] AspectJ. www.eclipse.org/aspectj.

[6] Cassandra JIRA. http://issues.apache.org/jira/browse/Cassandra.

[7] Hadoop JIRA Repository. http://issues.apache.org/jira/browse/Hadoop.

[8] HBase JIRA. http://issues.apache.org/jira/browse/HBase.

[9] HDFS Architecture Documentation. http://hadoop.apache.org/common/docs/current/hdfs_design.html.

[10] HDFS JIRA Repository. http://issues.apache.org/jira/browse/HDFS.

[11] ZooKeeper JIRA. http://issues.apache.org/jira/browse/ZooKeeper.

[12] Message 1800544: High Latency. http://communities.vmware.com, 2011.

[13] Personal Communication from Dhruba Borthakur of Facebook, 2011.

[14] Personal Communication from Andrew Baptist of Cleversafe, 2013.

[15] Personal Communication from Kevin Harms of Argonne National Laboratory, 2013.

[16] Personal Communication from Mike Kasick of CMU, 2013.

[17] Daniel Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, March 2009.

[18] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling Shared Scans of Large Data Files. In *VLDB '08*.

[19] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell C Sears. BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *EuroSys '10*.

[20] Amazon. Amazon S3 Availability Event. `http://status.aws.amazon.com/s3-20080720.html`, 2008.

[21] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: attack of the clones. In *NSDI '13*.

[22] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI '10*.

[23] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *SOSP '01*.

[24] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and Dave Patterson. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *SPDT '98*.

[25] Remzi H. Arpaci-Dusseau. Run-Time Adaptation in River. *ACM Transactions on Computer Systems (TOCS)*, 21(1):36–86, February 2003.

[26] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, Dave Patterson, and Kathy Yelick. Cluster I/O with River: Making the Fast Case Common. In *IOPADS '99*.

[27] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *HotOS VIII*, 2001.

[28] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *OSDI '12*.

[29] Algirdas A. Avižienis. The Methodology of N-Version Programming. In Michael R. Lyu, editor, *Software Fault Tolerance*, chapter 2. John Wiley & Sons Ltd., 1995.

[30] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07*.

[31] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08*.

[32] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *USENIX '09*.

[33] Luiz Barroso, Jeffrey Dean, and Urs Hoelzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[34] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[35] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI '06*.

[36] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[37] Ken Birman, Gregory Chockler, and Robbert van Renesse. Towards a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, June 2009.

[38] Dina Bitton and Jim Gray. Disk shadowing. In *VLDB '88*.

[39] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[40] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *ESA '06*.

[41] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005.

[42] Dhruba Borthakur. Hadoop avatarnode high availability. `http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html`, 2010.

[43] Bug Report. A block report processing may incorrectly cause the namenode to delete blocks. `https://issues.apache.org/jira/browse/HADOOP-1135`.

[44] Bug Report. A single slow (but not dead) map TaskTracker impedes MapReduce progress. `https://issues.apache.org/jira/browse/MAPREDUCE-562`.

[45] Bug Report. Corrupt replicas are not tracked correctly through block report from DataNode. `https://issues.apache.org/jira/browse/HDFS-900`.

[46] Bug Report. Decommissioning on NN restart can complete without blocks being replicated. `https://issues.apache.org/jira/browse/HDFS-3087`.

[47] Bug Report. HDFS should not remove blocks while in safemode. `https://issues.apache.org/jira/browse/HADOOP-3002`.

[48] Bug Report. Namenode accepts block report from dead datanodes. `https://issues.apache.org/jira/browse/HDFS-1250`.

[49] Bug Report. Reduce task stuck at 95.71% for a long time and the speculative execution does not kick in. `https://issues.apache.org/jira/browse/MAPREDUCE-92`.

[50] Bug Report. Shouldn't hold lock on rjob while localizing resources. `https://issues.apache.org/jira/browse/MAPREDUCE-2364`.

[51] Bug Report. TcpConnectionManager only ever has one connection. `https://issues.apache.org/jira/browse/CASSANDRA-488`.

[52] Bug Report. Using map output fetch failures to blacklist nodes is problematic. `https://issues.apache.org/jira/browse/MAPREDUCE-1800`.

[53] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems Export. In *OSDI '06*.

[54] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI '04*.

[55] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *OSDI '99*.

[56] Ronnie Chaiken, Bob Jenkins, Paul Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB '08*.

[57] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live - An Engineering Perspective. In *PODC '07*.

[58] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI '06*.

[59] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[60] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy H. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS '11*.

[61] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Anomaly? Application Change? Or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change. In *DSN '08*.

[62] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *SOSP '01*.

[63] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *SOSP '09*.

[64] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI '04*.

[65] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.

[66] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10*.

[67] Miguel Correia, Daniel Gomez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical hardening of crash-tolerant systems. In *USENIX ATC '12*.

[68] Paul Darga and Chandrasekhar Boyapati. Efficient Software Model Checking of Data Structure Properties. In *OOPSLA '06*.

[69] Jeffrey Dean. Underneath the Covers at Google: Current Systems and Future Directions. In *Google I/O '08*.

[70] Jeffrey Dean. Software Engineering Advice from Building Large-Scale Distributed Systems. `http://static.googleusercontent.com/media/research.google.com/en/us/people/jeff/stanford-295-talk.pdf`, 2011.

[71] Jeffrey Dean and Luiz Andre Barroso. Tail at Scale. *Communications of the ACM*, 2013.

[72] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04*.

[73] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP '07*.

[74] Michel Diaz. Modeling and analysis of communication and cooperation protocols using petri net based models. *Computer Networks (1976)*, 6(6):419–441, 1982.

[75] Forum Discussion. Weak Head. `http://forum.hddguru.com/search.php?keywords=weak-head`.

[76] Thanh Do. HDFS Append Bug Reports. `http://hdfswiki.pbworks.com/w/page/26821840/HDFS-Bug-Reports-For-Write-and-Append-Protocols`, 2010.

[77] Thanh Do and Haryadi S. Gunawi. The Case for Limping-Hardware Tolerant Clouds. In *HotCloud '13*.

[78] Thanh Do and Haryadi S. Gunawi. Impact of Limpware on HDFS: A Probabilistic Estimation. Technical Report TR-2013-08, Department of Computer Science, University of Chicago, July 2013.

[79] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *SoCC '13*.

[80] Erik Eckel. 10 tips for troubleshooting slowdowns in small business networks. `http://www.techrepublic.com`, 2007.

[81] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01*.

[82] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *VMCAI '04*.

[83] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281 –293, jun 2000.

[84] Michael Feldman. Startup Takes Aim at Performance-Killing Vibration in Datacenter. `http://www.hpcwire.com`, 2010.

[85] Daniel Ford, Franis Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlna. Availability in Globally Distributed Storage Systems. In *OSDI '10*.

[86] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *FAST '12*.

[87] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*.

[88] Garth Gibson. Reliability/Resilience Panel. In *HEC-FSIO '10*.

[89] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis and implications. In *SIGCOMM '11*.

[90] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[91] Robert L. Grossman and Yunhong Gu. On the Varieties of Clouds for Data Intensive Computing. *IEEE Data Engineering Bulletin*, 32(1):44–50, March 2009.

[92] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? In *ICSE '10*.

[93] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Usenix Security '09*.

[94] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST '09*.

[95] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl A. Waldspurger, and Mustafa Uysal. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *SoCC '11*.

[96] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI '11*.

[97] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Koushik Sen. Towards Automatically Checking Thousands of Failures with Micro-specifications. In *HotDep '10*.

[98] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *SOSP '07*.

[99] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *OSDI '08*.

[100] James Hamilton. On Designing and Deploying Internet-Scale Services. In *LISA '07*.

[101] James Hamilton. Observations on Errors, Corrections, and Trust of Dependent Systems. `http://perspectives.mvdirona.com`, 2011.

[102] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *OSDI '12*.

[103] Robin Harris. Bad, bad, bad vibrations. `http://www.zdnet.com`, 2010.

[104] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Wei Lin, Bing Su, Hongyi Wang, and Lidong Zhou. Wave Computing in the Cloud. In *HotOS XII*, 2009.

[105] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *FAST '09*.

[106] Gordon F. Hughes and Joseph F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives Using SATA Disk Drives. *ACM Transactions on Storage*, 1(1):95–107, February 2005.

[107] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC '10*.

[108] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. In *ASPLOS '02*.

[109] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2nd edition, July 2009.

[110] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and Detecting Real-World Performance Bugs. In *PLDI '12*.

[111] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[112] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-Box Problem Diagnosis in Parallel File Systems. In *FAST '10*.

[113] Lorenzo Keller, Paul Marinescu, and George Candea. AFEX: An Automated Fault Explorer for Faster System Testing. 2008.

[114] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI '07*.

[115] George Kola and Mary K Vernon. Target bandwidth sharing using endhost measures. *Performance Evaluation*, 64(9):948–964, 2007.

[116] Lars Michael Kristensen and Michael Westergaard. Automatic structure-based code generation from coloured petri nets: A proof of concept. In *Formal Methods for Industrial Critical Systems*, volume 6371 of *Lecture Notes in Computer Science*, pages 215–230. Springer Berlin / Heidelberg, 2010.

[117] Hairong Kuang, Konstantin Shvachko, Nicholas Sze, Sanjay Radia, and Robert Chansler. Append/Hflush/Read Design. `https://issues.apache.org/jira/secure/attachment/12445209/appendDesign3.pdf`.

[118] Sanjeev Kumar and Kai Li. Using model checking to debug device firmware. *ACM SIGOPS Operating Systems Review*, 36(SI):61–74, 2002.

[119] Sarah Kuranda. The 10 Biggest Cloud Outages Of 2013. `http://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm`.

[120] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *LADIS '09*.

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

[122] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. The Case for Drill-Ready Clouds. In submission to SoCC '14.

[123] Jiang Lin, Hongzhong Zheng, Zhichun Zhu, Eugene Gorbatov, Howard David, and Zhao Zhang. Software Thermal Management of DRAM Memory for Multicore Systems. In *SIGMETRICS '08*.

[124] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI '08*.

[125] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI '07*.

[126] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP '05*.

[127] Peter Lyman and Hal R. Varian. How Much Information? 2003. `www2.sims.berkeley.edu/research/projects/how-much-info-2003`.

[128] Om Malik. When the Cloud Fails: T-Mobile, Microsoft Lose Sidekick Customer Data. `http://gigaom.com`.

[129] Paul D. Marinescu, Radu Banabic, and George Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *USENIX ATC '10*.

[130] T.C. May and Murray H. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2–9, Jan 1979.

[131] Lucas Mearian. Facebook temporarily loses more than 10% of photos in hard drive failure. `www.computerworld.com`.

[132] Curt Monash. eBay's Two Enormous Data Warehouses. *DBMS2 (weblog)*, April 2009. `http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses`.

[133] Madanlal Musuvathi and Dawson R. Engler. Model Checking Large Network Protocol Implementations. In *NSDI '04*.

[134] E. Normand. *Nuclear Science, IEEE Transactions on*, 43(6):2742–2750, 1996.

[135] John Oates. Bank fined 3 millions pound sterling for data loss, still not taking it seriously. `www.theregister.co.uk`.

[136] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM J. Res. Dev.*, 40(1):41–50, 1996.

[137] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP '11*.

[138] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A logic-based network security analyzer. In *Usenix Security '05*.

[139] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio Lopez, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *SoCC '11*.

[140] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report CSD-02-1175, U.C. Berkeley, March 2002.

[141] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*.

[142] Carl Adam Petri. Communication with automata. 1966.

[143] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *FAST '07*.

[144] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. DryadInc: Reusing Work in Large-scale Computations. In *HotCloud '09*.

[145] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*.

[146] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. Pip: Detecting the unexpected in distrubted systems. In *NSDI '06*.

[147] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance. In *SOSP '01*.

[148] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *NSDI '11*.

[149] Stefan Savage and John Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *USENIX '96*.

[150] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST '07*.

[151] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *SIGMETRICS '09*.

[152] Simon CW See. Data Intensive Computing. In *Sun Preservation and Archiving Special Interest Group (PASIG '09)*, San Fancisco, California, October 2009.

[153] Anand Lal Shimpi. Intel Discovers Bug in 6-Series Chipset: Our Analysis. `http://www.anandtech.com`, 2011.

[154] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI '12*.

[155] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10*.

[156] Konstantin V. Shvachko. Hdfs scalability: The limits to growth. *;login:*, 35, 2010.

[157] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. In *EuroSys '06*.

[158] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST '04*.

[159] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *FAST '03*, pages 73–88.

[160] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05*.

[161] Brian Van Straalen and Phil Collela. Resiliency and codesign. In *DOE Exascale Research Conference*, 2012. `http://tinyurl.com/9wmq4cz`.

[162] Rajesh Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/ 0206tot_resiliency.html, February 2006.

[163] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. In *FAST '10*.

[164] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*.

[165] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI '04*.

[166] Alexander Szalay and Jim Gray. 2020 Computing: Science in an exponential world. *Nature*, (440):413–414, March 2006.

[167] Hadoop Team. Fault Injection framework: How to use it, test using artificial faults, and develop new faults. `http://hadoop.apache.org/docs/hdfs/r0.21.0/faultinject_framework.html`.

[168] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *FAST '13*.

[169] Wil MP van der Aalst. The application of petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998.

[170] Kashi Vishwanath and Nachi Nagappan. Characterizing Cloud Computing Hardware Reliability. In *SoCC '10*.

[171] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: performance insulation for shared storage servers. In *FAST '07*.

[172] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *SoCC '12*.

[173] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI '04*.

[174] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, volume 8, pages 281–294, 2008.

[175] Glenn Weinberg. The Solaris Dynamic File System. http://members.visi.net/∼thedave/sun/DynFS.pdf, 2004.

[176] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP '01*.

[177] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI '02*.

[178] Tom White. File Appends in HDFS. `http://www.cloudera.com/blog/2009/07/file-appends-in-hdfs`.

[179] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[180] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast Congestion Control for TCP. In *CoNEXT '10*.

[181] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *SOSP '09*.

[182] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed. Distributed Systems. In *NSDI '09*.

[183] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09*.

[184] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*.

[185] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*.

[186] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *FSE '11*.

[187] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI '08*.

[188] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *EuroSys '13*.

[189] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. 1979.

[190] Tao Zou, Guozhang Wang, Marcos Vaz Salles, David Bindel, Alan Demers, Johannes Gehrke, and Walker White. Making Time-stepped Applications Tick in the Cloud. In *SoCC '11*.