FATE and DESTINI: A Framework for Cloud Recovery Testing

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[*]

[†] University of California, Berkeley [‡] University of Wisconsin, Madison [*] Facebook

Abstract

As the cloud era begins and failures become commonplace, the fate and destiny of availability, reliability and performance are in the hands of failure recovery. Unfortunately, recovery problems still take place, causing downtimes, data loss, and many other problems. We propose 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. We have deployed our framework in three cloud systems (HDFS, ZooKeeper, and Cassandra), explored over 40,000 failure scenarios, wrote 74 specifications, found 16 new bugs, and reproduced 51 old bugs.*

1 Introduction

Large-scale computing and data storage systems, including clusters within Google [9], Amazon EC2 [1], and elsewhere, are becoming a dominant platform for an increasing variety of applications and services. These "cloud" systems are comprised of thousands of low-end machines (to take advantage of economies of scale [9, 15]) and thus require sophisticated and often complex distributed software to mask the underlying (and perhaps increasingly) poor reliability of commodity PCs, disks, and memories [4, 9, 16, 17].

A critical factor in the availability, reliability, and performance of cloud services is thus how they react to failure. Unfortunately, failure recovery has proven to be challenging in these systems. For example, in 2009, a large telecommunications provider reported a serious data-loss incident [26], and a similar incident occurred within a popular social-networking site [28]. Bug repositories of open-source cloud software hint at similar problems [2].

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 [6], whereas others believe that large-scale recovery remains underspecified [4]. These deficiencies leave us with 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 address this question, 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 [6, 34, 37], 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 uncovers 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 [14, 23, 24, 31, 38]. 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).

The rest of the paper is organized as follows. First, we dissect recovery problems in more detail (§2). Next, we define our concrete goals (§3), and present the design and implementation of FATE (§4) and DESTINI (§5). We then close with evaluations (§6) and conclusion (§7).

Problems	Count	Definitions and Examples
Incorrect	68	Recovery exists but it is still incorrect (examples are given throughout the paper).
Absent	14	Unanticipated failures (*e.g.*, corrupt metadata is not detected).
Coarse	7	A corrupt field causes a whole-machine shut down; A bad disk (out of many) shuts down a node.
Late	2	A failure not being detected/notified directly (*e.g.*, because of missing interrupt or wrong design).
Implications	**Count**	**Definitions and Examples**
Data loss	13	Unrecoverable data loss (*e.g.*, loss of metadata or blocks).
Unavailability	48	Inaccessible blocks/nodes, failed jobs/operations, prolonged timeouts/downtimes.
Corruption	19	Accessible data but the attributes/contents have been altered not as expected.
Unreliability	8	Reduced reliability (*e.g.*, a corrupt replica is not replaced with the other good replicas).
Performance	3	Increased latency or reduced bandwidth (*e.g.*, due to late recovery or under-replicated blocks)

Table 1: **Bug/Issue Study of Recovery Problems and Implications.**

2 Extended Motivation: 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 (§2.1). Since these stories do not reflect details, we study bug/issue reports of modern open-source cloud systems (§2.2). Finally, to get more insights, we dissect a failure recovery protocol (§2.3). We close this section by reviewing the state-of-the-art of testing (§2.4).

2.1 Lens #1: Practitioners' Experiences

As well-known practitioners and academics have stated: "the future is a world of failures everywhere" [11]; "reliability has to come from the software" [9]; "recovery must be a first-class operation" [8]. 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 [5]. In another paper, they reported another imperfect recovery that brought down the whole system [6]. After they tested Chubby with random multiple failures, they found more problems. BigTable engineers also stated that cloud systems see all kinds of failures (*e.g.*, crashes, bad disks, network partitions, corruptions, etc.) [7], which other practitioners also agree with [6, 9]. 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 [7]. To conclude, cloud systems face *frequent*, *multiple* and *diverse* failures [4, 6, 7, 9, 16]. Yet, recovery implementations are rarely tested with complex failures and are not rigorously specified [4, 6].

2.2 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 [18], Cassandra [22], HDFS [32]) publicly share in great detail real issues encountered in the field. Therefore, we performed an in-depth study of HDFS bug/issue reports [2]. 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. In total, there are 91 issues. Table 1 presents the variety of recovery problems that we found, including their significant implications.

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. In this regard, excluding our 5 submissions, 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. Reports from other cloud systems such as Cassandra and ZooKeeper also raise similar problems, implications, and observations.

2.3 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 essentially looks simple, but when different failures come into the picture, recovery complexity becomes evident. Figure 1 shows the write recovery protocol with three different failure scenarios. Throughout the paper, we will use HDFS terminology (*blocks*, *datanodes*/*nodes*, and *namenode*) [32] instead of GoogleFS terminology (chunks, chunk servers, and master) [10].

• **Data-Transfer Recovery:** Figure 1a shows a client contacting the namenode to get a list of datanodes to store three replicas of a block (`s0`). The client then ini-

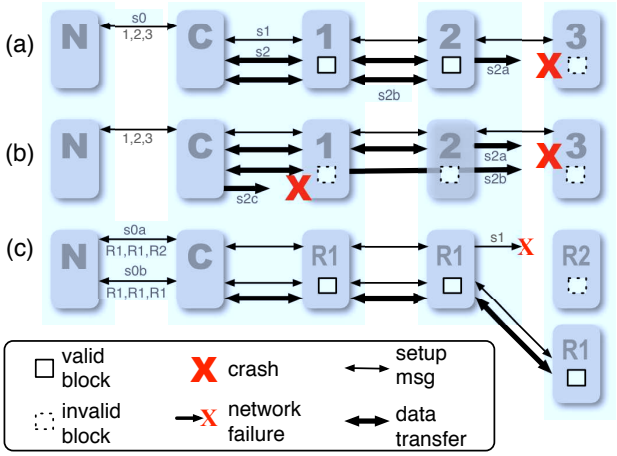

Figure 1: **HDFS Write Recovery Protocol.** *`N`, `C`, `R1/2`, and numeric letters represent the namenode, client, rack number, and datanodes respectively. The client always starts the activity to the namenode first before to the datanodes.*

tiates 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 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 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 (§6.2.1).

• **Setup-Stage Recovery:** Finally, Figure 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 [32].

• **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 §5.2.3).

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? [2]"

2.4 Current State of the Art: Does It Help?

In the last three sections, we presented our motivation for powerful testing frameworks for cloud systems. A natural question to ask is whether existing frameworks can help. We answer this question in two parts: failure exploration and system specifications.

2.4.1 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 variety of failures, one common practice is to inject a sequence of *random* failures as part of the unit test [6, 34].

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" [19, 27]. 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 [20, 36, 37, 38]. Modist, for example, is capa-

ble of exercising different combinations of failures (*e.g.*, crashes, network failures) [37]. As we discuss later, 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 [4, 9, 16, 17].

2.4.2 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 is to write extra checks 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 [38], 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 [14]. 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). As we will see later, specifying recovery requires "behavioral" specifications.

Another advanced checking approach is WiDS [23, 24, 37]. 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 [23, 24]. 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 [23].

Frameworks for declarative specifications exist (*e.g.*, Pip [31], P2 Monitor [33]). P2 Monitor only works if the target system is written in the same language [33]. Pip facilitates declarative checks, but a check is still written in over 40 lines on average [31]. 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 [2] – hence the number of written specifications is typically small and does not scale to the complexity of the system.

3 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 [13]. However, our framework is generic for other distributed systems. Our targets so far are HDFS [32], ZooKeeper [18] and Cassandra [22]. We mainly use HDFS as our example in the paper. 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 (*e.g.*, 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. To realize this, our framework must meet two requirements. First, the specifications must be developer-friendly (*i.e.*, concise, fast to write, yet easy to understand). Otherwise, 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 because 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.

4 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 (§4.1), a ready-to-use failure service which can be integrated seamlessly to cloud systems (§4.2), and novel failure prioritization strategies that speed up testing time by an order of magnitude (§4.3).

4.1 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 2). Below we describe these subcomponents in more detail.

• **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, 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 are 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 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.

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 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. For space, some fields are not shown.*

4.2 Architecture

We built FATE with an aim towards quick and seamless integration to our target systems. Figure 2 depicts the four components of FATE: workload driver, failure surface, failure server, and filters.

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

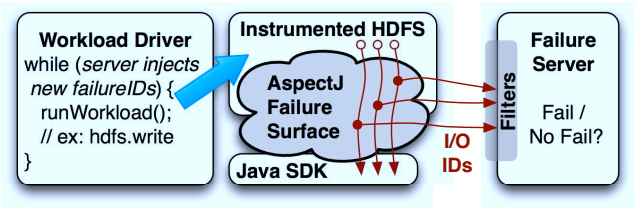

Figure 2: **FATE Architecture.**

4.2.2 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 apriori. 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`).

4.2.3 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 any failures only at datanodes.

4.3 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. 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. Due to space constraints, we could not show the detailed pseudo-code, and thus we only present the algorithms at a high-level. We will evaluate the algorithms in Section 6.3. We also emphasize that our proposed strategies are built on top of the information carried in failure IDs, and hence display the power of failure IDs abstraction.

4.3.1 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 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 a strategy that we call *recovery-behavior clustering*. The goal is 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

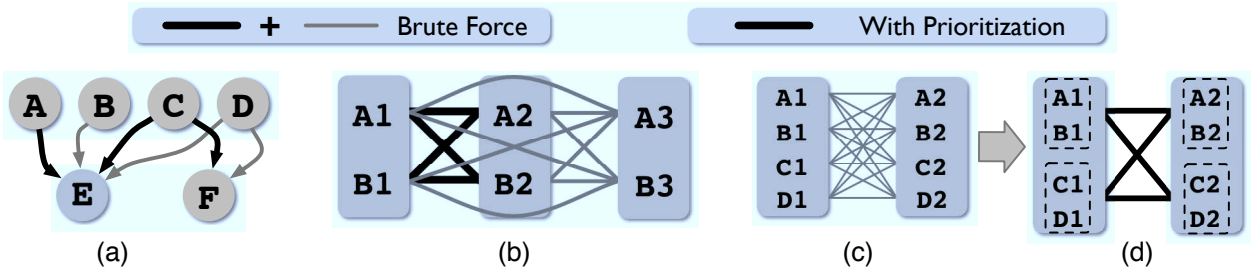

Figure 3: **Prioritization of Pairwise Dependent and Independent Failures.**

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 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 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.

4.3.2 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 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 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 3c. Here, instead of having two static failure points, there are four, which leads 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 3d). In practice, we have seen a reduction from fifteen SFP to eight $SFP_{clustered}$.

4.4 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 §6.3). If developers have the time and resource, they could fall back to brute-force mode for more confidence. So far, we have only explained our algorithms for two-failure scenarios. We have generalized them to three-failure, but cannot present them due to space constraint.

5 DESTINI: Declarative Testing Specifications

After failures are injected, developers still need to verify system correctness. As described in the motivation (§2.4), DESTINI attempts to improve the state-of-the-art of writing system specifications. In the following sections, we first describe the architecture (§5.1), then present some examples (§5.2), and finally summarize the advantages (§5.3). Currently, we target recovery bugs that reduce availability (*e.g.*, unmasked failures, fail-stop) and reliability (*e.g.*, data-loss, inconsistency). We leave performance and scalability bugs for future work.

5.1 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 [3, 25] and for verifying some aspects of system correctness (*e.g.*, security [12, 30]). 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 [3].

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 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.

5.1.1 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 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).

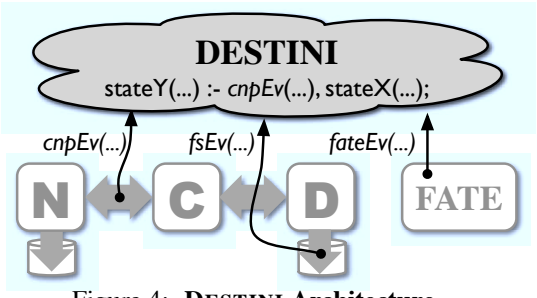

Figure 4: **DESTINI Architecture.**

5.2 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 (§5.2.1). To help complex debugging process, the second example shows how developers can incrementally add tighter specifications (§5.2.2). The third example presents specifications that incorporate a different type of failure than the first two examples (§5.2.3). Finally, we illustrate how developers can refine existing specifications (§5.2.4).

5.2.1 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 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

	Section 5.2.1		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);

	Section 5.2.2		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;

	Section 5.2.3		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);

	Section 5.2.4		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: **Sample Specifications.** *The table lists all the rules we wrote to specify the problems in Section 5.2; Rules* aX, bX, cX, *and* dX *are for Sections 5.2.1, 5.2.2, 5.2.3, and 5.2.4 respectively. 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). (**) Rule* d2 *is refined in* d3 *to* d5*; these rules are described more in our short paper [13].*

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 can 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**).

5.2.2 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 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 1b in §2.3). 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** 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

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 errorneous 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 4: **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 Table 3 are populated.*

views (*e.g.*, global, client and datanode as shown in **a1**, **b1**, **b4** respectively). Table 4 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.

5.2.3 Specifying Rack-Aware Replication Policy

In this example, we write specifications for HDFS rack-aware replication policy, an important policy for high availability [10, 32]. Unlike previous examples, this example incorporates network partitioning failure mode.

According to the HDFS architects [32], 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.

With these checks, DESTINI found the bug in Figure 1c (§2.3), 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.

5.2.4 Refining Specifications

In the second example (§5.2.2), we demonstrated how developers can *incrementally add* detailed specifications. In this section, we briefly show how developers can *refine* existing specifications (an extensive description can be found in our short paper [13]).

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.

5.3 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.

6 Evaluation

We evaluate FATE and DESTINI in several aspects: the general usability for cloud systems (§6.1), the ability to catch multiple-failure bugs (§6.2), the efficiency of our prioritization strategies (§6.3), the number of specifications we have written and their reusability (§6.4), the number of new bugs we have found and old bugs reproduced (§6.5), and the implementation complexity (§6.6).

6.1 Target Systems and Protocols

We have integrated FATE and DESTINI to 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).

6.2 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.

6.2.1 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 [35]. HDFS developers implement append with a custom protocol; their latest append design was written in a 19-page

document of prose specifications [21]. Append was finally supported after being a top user demand for three years [35]. As a note, Google FS also supports append, but its authors did not share their internal design [10].

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 [2].

6.2.2 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 [2]. Moreover, data-loss bugs are serious ones [26, 28, 29].

6.3 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 strategies for multiple-failure prioritization.

Workload	#F	STR	#EXP	FAIL	BUGS
Append	2	BF	**1199**	116	**3**
		PR	**112**	17	**3**
Append	3	BF	**7720**	**3693	*3
		PR	**618**	72	*3
Write	2	BF	**524**	120	**2**
		PR	**49**	27	**2**
Write	3	BF	**3221**	911	*2
		PR	**333**	82	*2

Table 5: **Prioritization Efficiency.** *The columns from left to right are the number of injected failures per run (F), exploration strategy (STR), combinations/experiments (EXP), failed experiments (FAIL), and bugs found (BUGS). BF and PR stands for brute-force and prioritization respectively. 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). (*) 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 (not shown).*

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 5 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.

6.4 Specifications

In the last six months, we have written 74 checks on top of 174 rules for a total of 351 lines (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 (§5.2.1 in Table 3) is 16:1, but the ratio now is 3:1.

Table 6 compares DESTINI with other related work. 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

Type	Framework	#Chks	Lines/Chk
S/I	D3S [23]	10	**53**
D/I	Pip [31]	44	**43**
S/I	WiDS [24]	15	**22**
D/D	P2 Monitor [33]	11	**12**
D/I	DESTINI	74	**5**

Table 6: **DESTINI vs. Related Work.** *The table compares* DESTINI *with related work. D, S, and I represent declarative, scripting, and imperative languages respectively. X/Y implies specifications in X language for systems in Y language. We divide existing work into three classes (S/I, D/D, D/I).*

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.

6.4.1 ZooKeeper

We have integrated our framework to ZooKeeper [18]. 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.

6.4.2 Cassandra

We have also done the same for Cassandra [22], 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.

6.5 New Bugs and Old Bugs Reproduced

We have tested HDFS for over eight months and submitted 16 new bugs, 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. For Cassandra and ZooKeeper, we just began integrating our framework to these systems roughly two months ago. Recently, we observed some failed experiments, but since we do not have the chance to debug all of them, we have 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 (§2.2) and selected 74 that relate to our target workloads (§6.1). 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.

6.6 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 (§5.1). 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.

7 Conclusion and Future Work

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. For example, one new approach we are currently investigating is the increased use of *pessimism* to avoid problems during recovery. For example, HDFS lease recovery would have been more robust had it not trusted aspects of the append protocol to function correctly (§6.2). Many other examples exist; only through further careful testing and analysis will the next generation of cloud systems meet their demands.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.

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

[3] 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*.

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

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

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

[7] 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*, pages 205–218.

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

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

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*, pages 29–43.

[11] Garth Gibson. Reliability/Resilience Panel. In *HEC-FSIO '10*.

[12] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Usenix Security '09*.

[13] 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*.

[14] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *OSDI '08*.

[15] James Hamilton. Cloud Computing Economies of Scale. In *MIX '10*.

[16] James Hamilton. On Designing and Deploying Internet-Scale Services. In *LISA '07*.

[17] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *FAST '09*.

[18] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX '10*.

[19] Lorenzo Keller, Paul Marinescu, and George Candea. AFEX: An Automated Fault Explorer for Faster System Testing. 2008.

[20] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI '07*.

[21] 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.

[22] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *LADIS '09*.

[23] 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*.

[24] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI '07*.

[25] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP '05*.

[26] Om Malik. When the Cloud Fails: T-Mobile, Microsoft Lose Sidekick Customer Data. http://gigaom.com.

[27] Paul D. Marinescu, Radu Banabic, and George Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *USENIX '10*.

[28] Lucas Mearian. Facebook temporarily loses more than 10% of photos in hard drive failure. www.computerworld.com.

[29] John Oates. Bank fined 3 millions pound sterling for data loss, still not taking it seriously. www.theregister.co.uk.

[30] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A logic-based network security analyzer. In *Usenix Security '05*.

[31] 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*.

[32] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10*.

[33] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. In *EuroSys '06*.

[34] Hadoop Team. Fault Injection framework: How to use it, test using artificial faults, and develop new faults. http://issues.apache.org.

[35] Tom White. File Appends in HDFS. http://www.cloudera.com/blog/2009/07/file-appends-in-hdfs.

[36] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed. Distributed Systems. In *NSDI '09*.

[37] 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*.

[38] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*.