Towards Automatically Checking Thousands of Failures with Micro-specifications

Haryadi S. Gunawi, Thanh Do†, Pallavi Joshi, Joseph M. Hellerstein,
Andrea C. Arpaci-Dusseau†, Remzi H. Arpaci-Dusseau†, and Koushik Sen

University of California, Berkeley † University of Wisconsin, Madison

Abstract

Recent data-loss incidents have shown that existing large distributed systems are still vulnerable to failures. To improve the situation, we propose two new testing approaches: failure testing service (FTS) and declarative testing specification (DTS). FTS enables us to systematically push a system into thousands of failure scenarios, leading us to many critical recovery bugs. With DTS, we introduce "micro-specifications", clear and concise specifications written in Datalog style, which enables developers to easily write, refine, and manage potentially hundreds of specifications.

1 Introduction

The power of large clusters behind cloud computing has brought us not only benefits but also a new challenge: a growing number and frequency of failures that must be managed [8, 9]. Failing to deal with failures will directly impact the reliability and availability of data and jobs. Unfortunately, recent data-loss incidents experienced by a telecommunication provider [14], a popular social networking site [15], and a large bank [16] still display the vulnerability of existing systems to hardware failures, failures such as machine crashes, disk and network failures. This leaves us with an important question: How can we verify the correctness of large distributed systems in dealing with a growing number failures?

We believe a proper answer requires two more rigorous approaches than the current state-of-the-art of testing. First, we must begin with a new approach that can *systematically* push a system into many possible failure scenarios. Some existing tools are often not equipped with a fault-injection feature, and thus failures are injected manually. Some others have the feature, but only inject failures of the same type (*e.g.*, crashes) [20]. When it comes to injecting a wide variety of failures, the state-of-the-art in industry is to do it randomly [3, 11, 19].

Second, after thousands of failures are systematically injected, we still need to verify the correctness of many properties of the system. For a large distributed system, there are potentially hundreds of properties that must be checked (especially under failures). However, with existing approaches, a check sometimes has to be written in tens of lines of code (*e.g.*, in C++ [20], or a scripting language [12]). The drawbacks are two-fold. First, they hinder developers from writing a large number of specifications; in practice, the number of deployed checks for a new system is typically small, and hence does not scale to the complexity of the system. Second, even if we have hundreds of specifications (*e.g.*, in old systems where the developers have incrementally added them over the years), they could be as big as the system code itself. As a result, the specifications are also likely to be wrong and correcting them is not always straightforward [7]. Further complicating the matter, the specifications must also evolve as the system evolves [11].

To overcome the first challenge above, we present *failure testing service* (FTS), a framework that can systematically exercise many combinations of varieties of failures (and by "many" our target is "thousands of scenarios"). The key ingredient to a systematic fault-injection technique is to identify failure points and a list of possible failures that could happen at a particular failure point (crashes, disk/network faults, etc.). A failure point and an injected failure constitute a *failure ID* that FTS will explore; exercising a combination of failures is essentially exercising a combination of failure IDs.

To enable developers to write specifications quickly and incrementally, we introduce *declarative testing specification* (DTS). At the heart of DTS is a declarative relational logic language (based on Datalog) that enables a check to be written in only a few lines. We chose Datalog style due to its nature of compactness and expressiveness in specifying logical relationships [1, 6, 13, 17]. With DTS, we promote a practical iterative style of writing specification: a developer can begin with a handful of high-level specifications, and as she finds new bugs, she can add more detailed specifications to precisely pinpoint the bugs. If the same bugs appear again in the future (as the code being developed), the tighter specifications will allow the developer to avoid wasting another hours of debugging time.

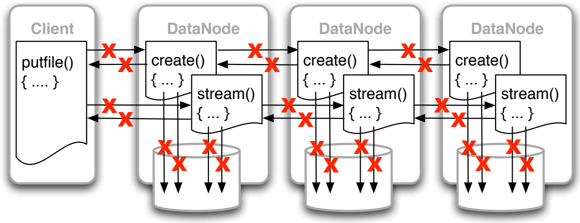

Figure 1: **HDFS Write Protocol.** *The figure illustrates when and where failures could occur in this protocol. Communications to the master node is excluded for brevity.*

Info Type	Field		Example value
Static	Func. call	:	OutputStream.flush()
	I/O type	:	Write
	Location	:	BlockReceiver.java
	Line	:	45
Dynamic	Stack trace	:	(the stack trace)
Domain specific	Thread name	:	Block Receiver
	Target I/O	:	Disk1, Metafile
FTS	Failure	:	Bad disk
	Hash Value	:	*1289065658*

Table 1: **A Sample of Failure ID.** *A failure ID contains static, dynamic, and domain-specific information about a failure point and an injected failure. Hash is used to label a failure ID.*

This paper makes the three following contributions:

- We introduce failure IDs as a new abstraction for exploring failures systematically. In addition, we also provide a ready-to-use "failure surface" for Java-based systems (§2).
- With DTS, we introduce "micro-specifications", clear and concise specifications written in Datalog style, which enables developers to manage hundreds of specifications. We also present a design pattern for declarative debugging, that is, how a developer could refine specifications iteratively (§3).
- We have applied FTS and DTS to the Hadoop File System (HDFS), an open-source version of the Google File System [5], and uncovered many critical recovery bugs. We chose HDFS as our first target due to its complexity (over 25 KLOC) and its growing popularity (it has been widely deployed in over 80 medium to large organizations including Amazon, Yahoo, and Facebook).

After presenting our contributions, we follow with a discussion of new challenges (§4) and close with related work (§5) and conclusion (§6).

2 Failure Testing Service

Figure 1 motivates the need for a systematic failure service. For example, in a distributed write, there are many points where the system components could fail (labeled with X). One form of expectation is that the write protocol should succeed if at least one node is alive [5]. The developers might want to verify that this specification holds even if, for example, the disk at the second node fails in the create phase; or, if the first node *and* the second node crash in the data streaming phase.

To help developers in this regard, the ultimate goal of FTS 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 in model-checking is to record the states that have been explored, which is commonly done by hashing the abstract state of the system and recording the history of the hashes. Similarly in our case, we need a clear abstract representation of a failure, which can be hashed and thus recorded in the failure history. Below, we first introduce the concept of failure IDs and follow with the overall framework. So far, we have used FTS to exercise more than 40,000 combinations of failures.

2.1 Failure IDs

To construct a failure ID, we first define a *failure point*, a system/library call that performs disk or network I/O. For every failure point, FTS generates a list of possible failures that could happen before and after. For example, FTS could throw an exception before a disk-write failure point. FTS could also crash a node after the node receives a message but before it sends an acknowledgment. When FTS exercises a failure ID (*i.e.*, injects a particular failure at a particular failure point), FTS records the hash value of the failure ID in the failure history.

Table 1 shows an example of a failure ID. The table also shows that a failure point contains more complex information (static, dynamic, and domain-specific) than what we have described above. These information are essential to increase failure coverage. For example, if a static failure point could be called by different upper-level functions, then dynamic information such as stack trace is useful to expand the failure scenarios. Domain-specific information such as target I/O is valuable because a common function could write to different file types or send messages to different nodes.

2.2 Implementation

We plan to integrate the concept of failure IDs to a model checker (since a failure ID does not contain the system state). However, since not all systems support a ready-to-use model-checker, we build our own testing framework

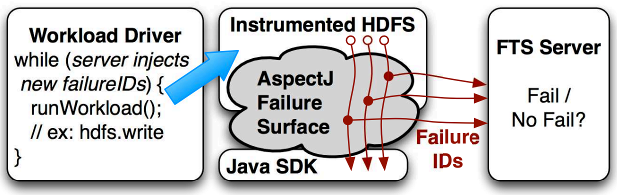

Figure 2: **FTS Framework.**

that could be quickly integrated to the system we want to test. Figure 2 depicts the overall framework of FTS. We first instrument the system (*e.g.*, HDFS) by inserting a "failure surface" which builds failure IDs and send them to the FTS server, which then makes failure decisions based on the failure history. The developer attaches the workload to be tested at the workload driver and specifies the maximum number of failures (MAX). The while loop exits when the server does not see a new combination of MAX failure IDs.

FTS is written in 4180 lines of code in Java. We use aspect oriented programming (AspectJ) to insert the failure surface; no changes to the system under test. As of now, FTS is able to inject transient failures, persistent disk failures, crashes, and disk corruption. In the future, we will support other types of failures (*e.g.*, delays to emulate message reordering).

2.3 Preliminary Results

We have run FTS on three HDFS workloads: write, append and master reboot. In total, FTS has generated 394 unique failure IDs and exercised 41332 unique combinations (with a maximum of three failures per run), out of which 22872 result in unsuccessful experiments. The ability to explore this large number of combinations has led us to numerous critical recovery bugs (more in Section 3.4). Furthermore, we found some sections of code that would not have been executed unless three failures are injected. Thus, with FTS, the vision of "towards 100% coverage of recovery code" becomes feasible.

To show that FTS can be deployed to other systems quickly, we have ported FTS to two different versions of the Apache Lucene concurrency library in just a few hours. In this particular experiment, we would like to find new concurrency bugs in the presence of failures (*e.g.*, a hang bug because an exception block does not wake up waiting threads); we note that many novel concurrency bug-finding tools often do not incorporate failures. We have run 23 workloads (out of the available 184 workloads in their JUnit tests) and found 4 concurrency bugs (2 are new).

3 Declarative Testing Specification

After failures are injected, the developer needs to verify the system correctness. This can be done in many ways (*e.g.*, via external behaviors or detailed internal checks). Typically, a single check could reach one hundred lines of code [7, 11, 19]. As a result, managing hundreds of checks could be complicated (not to mention that they must evolve as the system evolves [11]).

With DTS, we attempt to improve the state-of-the-art of writing testing specification. To achieve this, we explored writing "micro-specifications", clear and concise specifications written in Datalog style. We chose this style as it has been successfully used in building distributed systems declaratively [1, 13] and in verifying some aspects of system correctness (*e.g.*, security [6, 17]). However, since the innovation of high-performance declarative language is still underway [1], we feel that using Datalog for just writing system specifications is a sweet spot to explore.

In the next two sections, we show how micro-specifications promote a practical iterative style of writing specifications. More specifically, the first example shows how a developer can *refine* loose specifications into tighter ones, while the second example illustrates how to *incrementally add* more detailed specifications. The motivation for these examples is that a developer never begins with complete and precise specifications. But, as she unearths new problems, she might wish to refine existing specifications or add more specifications to pinpoint similar problems quickly in the future.

3.1 Refining Specifications

This section focuses on demonstrating how a developer could refine a high-level specification into a tighter one. Here, we use the HDFS log recovery process as an example. We begin with a high-level specification that will catch data-loss bugs:

```
lostFiles(F) :- userFiles(F), not-in server(F)
```

The specification is written as a Datalog rule which consists of a head (`lostFiles`) and predicates in the body (`userFiles` and `server`). The head is evaluated when the body is true. A comma between predicates represents conjunction. Thus, the rule specifies the relation: "a user file `F` is lost if it does not exist at the server."

Next, we need to "fill" the test predicates with runtime facts; the head and the predicates are essentially database tables. The `userFiles` table is filled whenever the HDFS client write API returns a success. Populating the `server` table is a bit more complicated. As a background, HDFS maintains two server files that store the metadata of user files: the image file (`img`) and the log file (`log`). When a client stores a file, it is first recorded in the log file. Upon

Contents of {img, log, log2, img2}	Steps (1 to 5)	User files (f_1 and f_2) in:
f_1, f_2, $-$, $-$	1. Start	img + log
f_1, f_2, $\emptyset$, $-$	2. Create an empty log2	img + log
f_1, f_2, $\emptyset$, f_1f_2	3. Merge img and log to img2	img + log
f_1, $\emptyset$, $-$, f_1f_2	4. Rename log2 to log	img2
f_1f_2, $\emptyset$, $-$, $-$	5. Rename img2 to img	img + log

Table 2: **Log Recovery Protocol.** *The table shows the process of merging user-file metadata (f_1 and f_2) in* img *and* log. *If a crash occurs in the middle, the next reboot will start at a different step depending on which of the four files exist (not shown). $\emptyset$ and $-$ represent empty and non-existent file respectively.*

a master reboot (or periodically), the log recovery process will merge the two files into the image file and empty the log file. To ensure idempotency, HDFS utilizes two other files (log2 and img2). In short, *at any time*, user files should exist in the union of all the four files. Thus, we can simply write the specification below, which reads: "a user file F exists at the server if F exists in any of the four server files." Uppercase and lowercase letters (*e.g.*, F and img) stand for variables and constants respectively.

```
server(F) :- filesIn(F, img);
server(F) :- filesIn(F, log);
server(F) :- filesIn(F, log2);
server(F) :- filesIn(F, img2);
```

We use FTS to automatically insert all possible crashes within this process. Interestingly, the rules above trigger a violation of the data-loss specification. However, since the rules are not rigorous enough, we were not able to exactly pinpoint the bug. We spent a couple hours debugging, and then refined the specification (and hence the iterative process of writing specifications).

The new specification reflects in detail the five-step process shown in Table 2. That is, depending on the progress, user files are expected to be in a different subset of the four files as shown in the last column of the table. For example, during the process of merging img and log to img2 at step 3, user files are expected to be in img and log, but not in img2, because img2 is still not complete. However, when log is emptied at step 4, img2 is complete, and thus user files should be found in img2, but not in img and log. The way HDFS keeps track the steps is via the existence of the four files (column 1). To express this, we simply introduce two new relational tables exists(img,log,log2,img2) and step(Num):

```
step(1) :- exists(1, 1, 0, 0);
step(2) :- exists(1, 1, 1, 0);
step(3) :- exists(1, 1, 1, 1);
step(4) :- exists(1, 1, 0, 1);
```

Using the new relations above, we can simply write new server rules that correspond to the logic in the last column of Table 2, that is, "user files should be in img2 at step 4, or in img and log otherwise":

```
server(F) :- filesIn(F, img2), step(N), N == 4;
server(F) :- filesIn(F, img) , step(N), N != 4;
server(F) :- filesIn(F, log) , step(N), N != 4;
```

With this more rigorous specification, we were able to pinpoint the data-loss bug. As a background, if the master node crashes exactly before the log renaming operation at step 4, the protocol will begin again from step 1 (not shown). However, exactly before step 4, there is a bug that truncates the log file (f_2 is no longer in log). Thus, if a crash happens after the truncate bug and before step 4, the next reboot will merge img (f_1) with an empty log! The final img file will only contain f_1, and hence f_2 is lost. This bug still exists in the latest released version of HDFS.

The lostFiles rule captures the bug because all rules that depend on log are re-evaluated when the bug truncates log: file f_2 is removed from filesIn(F, log), which will then remove f_2 from server(F) (because the bug is at step 3). Finally, f_2 is added to lostFiles because f_2 is in userFiles but not in server. In summary, in this section, we have written a full-page prose specification of the log recovery process, and it has been elegantly summarized in just 8 lines of Datalog rules.

3.2 Adding Specifications Incrementally

To illustrate the generality of the iterative process above, we briefly give another example where we catch unavailability bugs. This time, rather than refining a specification, we incrementally add more specifications to pinpoint the bugs. We begin with a general specification below which reads: "there is an unavailability bug if a write operation fails but there is at least one good datanode."

```
unavailability(F, Num) :-
 write_op(F, fail), good_nodes(Num), Num >= 1;
```

We caught some bugs that break the rule. One bug is a buggy failover where the master keeps returning the same datanodes that the client cannot reach. This is because it takes the master 10 minutes to detect a dead datanode by default. But, since the master does not incorporate the client's view of unreachable datanodes, the master does not give the client another set of available datanodes. To catch this specific design bug, we simply added a new rule:

```
bad_failover(BadNode) :-
 master_returns(BadNode), unreachable_nodes(BadNode);
```

3.3 Implementation

To fill the rules with runtime facts, one can write some form of state-exposer that interposes the internal functions of each node [11]. However, we decided to interpose and reverse-engineer the system API and the disk and network protocols, primarily because inter-node protocols and on-disk formats change less frequently than the internal functions. The obtained information is then "translated" into Datalog. For example, when a buffer is written to the `img` file, we extract the file entries in the buffer, and add them to `filesIn(F, img)`; when the HDFS write API returns a success, the corresponding file is added to `userFiles(F)`; when HDFS creates and deletes any of the four storage files, the `exists()` relation is re-evaluated. After the translation phase, the tester enters the declarative world in which she can build the abstract model of the system succinctly. The DTS translation mechanism for HDFS is written in 1200 lines of code in Java.

3.4 Preliminary Results

So far, we have written 50 rules in less than 70 lines. The number is still small because we have been writing specifications in a bug-driven manner, that is, for every bug that we found (from external observation), we wrote more specifications to pinpoint the bug more precisely. We believe that there is nothing that prevents us from writing many more specifications, which is our next top priority.

We are still in the process of debugging the thousands of unsuccessful experiments reported in Section 2.3. As of now, we have uncovered 21 bugs (15 are new, some of which have been confirmed by the HDFS developers) that lead to unavailability, data loss, and inconsistencies. Since there is not enough space to explain all of them, we only present our high-level observations below.

First, unsurprisingly, a system implementation is much more complex than the prose specification; when we read the simple prose specification of the HDFS write protocol, we had little confidence that we would find new bugs. But, we were wrong; we found many bad failover strategies that fail the corresponding operations even though good resources are still available. Second, replicas are not always valuable; they only prevent a single point of hardware failure, but software is still the single point of failure. A prime example is the data-loss bug described in Section 3.1; the bug causes the master node to lose all replicas. Another example is some bugs in the append protocol that make the datanodes lose all replicas of the affected files.

3.5 Summary

Specifications written in DTS-style are generally short and easy to understand. From our experience, the major time spent is in understanding the HDFS internal designs. Once we understand them, writing the specifications can be done in a relatively short amount of time. However, we expect that, if the developers adopt our approach, they can write specifications more and faster than us. Another major effort in building DTS is in building the translation mechanism (*e.g.*, converting information available from the Java implementation to Datalog events). More specifically, this needs to be done for all inter-node and node-disk interfaces. If these interfaces change, our mechanism must change too. But again, we expect that interfaces change rarely. If the system's internal designs change but the interfaces don't, then we only need to change the specifications in DTS, which is again considerably straightforward given the short amount of lines of code that we need to deal with.

4 Future Work and Discussion

We have begun deploying FTS and DTS to two other large distributed systems: Cassandra (an open-source version of Amazon Dynamo and Google BigTable) and ZooKeeper (an open-source version of Google Chubby). Also, due to our initial success, the developers of these systems have shown interest in adopting our approaches to their systems.

Apart from this initial success, there are new challenges ahead. First, we need to intelligently prune the failure points; the number of combinations of failures grows exponentially (*e.g.*, in a particular setup with 149 failure IDs and 3 injected failures per experiment, FTS generates 17263 thousands of experiments in over five hours). This prohibits us to explore more failures per experiment. We have sketched out some pruning techniques, but unfortunately they are out of the scope of this paper.

Second, we need some form of heuristics to classify unsuccessful experiments to the actual bugs. From our experience, thousands of unsuccessful experiments map to only a few number of bugs (§2.3 and §3.4). Without heuristics, we have to debug one experiment at a time.

5 Related Work

We have compared the resemblance between FTS and model-checkers [10, 19, 20]. To have a model-checker exercise failures systematically, the notion of failure IDs can be directly employed. We also have compared DTS with the state-of-the-art of writing testing specifications [11, 12, 20] and shown that DTS enables more concise specifications. Other than these, there are some other relevant work: D3 uses a variant of Datalog to write checks on top of distributed log messages [4], while ours run on top of disk and network communications; Singth *et* *al.* also use a variant of Datalog, but their checks only run on systems built in the same language [18]; SQCK proposes rewriting local file system checkers with SQL-style declarative queries, while DTS targets distributed systems and proposes a more compact and elegant Datalog-style approach. Finally, Cloud9 also plans to introduce a new testing specification that is accessible to programmers [2].

6 Conclusion

As failure becomes the norm, when data is lost or availability is reduced, we should no longer blame the failure, but rather the inability to recover from the failure. Thus, we began with a simple question: How can we assess the quality of existing systems in handling thousands of possible failures? This simple question gave birth to FTS. As we achieve this end, we were confronted with the fact that, for a large distributed system, there are a large number of properties that need to be verified (not just state invariants, but also specific behaviors of the recovery protocols). This gave birth to DTS, which has enabled us to write, refine, and manage many micro-specifications easily; the more specifications we can write, the more properties we can verify, the finer bug reports we will produce, and the less time we will spend on debugging.

7 Acknowledgments

We thank Peter Alvaro and the anonymous reviewers for their feedback and comments. We also thank members of the BOOM research group, in particular Tyson Condie and Neil Conway who built the Java-Overlog runtime. This material is based upon work supported by the National Science Foundation under grant Nos. IIS-0713661, CNS-0722077 and IIS-0803690, the Air Force Office of Scientific Research under Grant No. FA95500810352, and gifts from Microsoft, IBM and Yahoo!. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] 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*.

[2] George Candea, Stefan Bucur, and Cristian Zamfir. Automated Software Testing as a Service. In *SOCC '10*.

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

[4] Byung-Gon Chun, Kuang Chen, Gunho Lee, Randy H. Katz, and Scott Shenker. D3: Declarative Distributed Debugging. UC. Berkeley Technical Report No. UCB/EECS-2008-27, 2008.

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

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

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

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

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

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

[11] 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*.

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

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

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

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

[16] John Oates. Bank fined 3 millions pound sterling for data loss, still not taking it seriously. `www.theregister.co.uk/2009/07/22/fsa_hsbc_data_loss`.

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

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

[19] 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*.

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