Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, University of Toronto

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.
October 6–8, 2014 • Broomfield, CO
978-1-931971-16-4

Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.
Simple Testing Can Prevent Most Critical Failures
An Analysis of Production Failures in Distributed Data-intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, Michael Stumm
University of Toronto

Abstract
Large, production quality distributed systems still fail periodically, and do so sometimes catastrophically, where most or all users experience an outage or data loss. We present the result of a comprehensive study investigating 198 randomly selected, user-reported failures that occurred on Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis, with the goal of understanding how one or multiple faults eventually evolve into a user-visible failure. We found that from a testing point of view, almost all failures require only 3 or fewer nodes to reproduce, which is good news considering that these services typically run on a very large number of nodes. However, multiple inputs are needed to trigger the failures with the order between them being important. Finally, we found the error logs of these systems typically contain sufficient data on both the errors and the input events that triggered the failure, enabling the diagnose and the reproduction of the production failures.

We found the majority of catastrophic failures could easily have been prevented by performing simple testing on error handling code – the last line of defense – even without an understanding of the software design. We extracted three simple rules from the bugs that have lead to some of the catastrophic failures, and developed a static checker, Aspirator, capable of locating these bugs. Over 30% of the catastrophic failures would have been prevented had Aspirator been used and the identified bugs fixed. Running Aspirator on the code of 9 distributed systems located 143 bugs and bad practices that have been fixed or confirmed by the developers.

1 Introduction
Real-world distributed systems inevitably experience outages. For example, an outage to Amazon Web Services in 2011 brought down Reddit, Quora, FourSquare, part of the New York Times website, and about 70 other sites [1], and an outage of Google in 2013 brought down Internet traffic by 40% [21]. In another incident, a DNS error dropped Sweden off the Internet, where every URL in the .se domain became unmappable [46].

Given that many of these systems were designed to be highly available, generally developed using good software engineering practices, and intensely tested, this raises the questions of why these systems still experience failures and what can be done to increase their resiliency. To help answer these questions, we studied 198 randomly sampled, user-reported failures of five data-intensive distributed systems that were designed to tolerate component failures and are widely used in production environments. The specific systems we considered were Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis.

Our goal is to better understand the specific failure manifestation sequences that occurred in these systems in order to identify opportunities for improving their availability and resiliency. Specifically, we want to better understand how one or multiple errors evolve into component failures and how some of them eventually evolve into service-wide catastrophic failures. Individual elements of the failure sequence have previously been studied in isolation, including root causes categorizations [33, 52, 50, 56], different types of causes including misconfigurations [43, 66, 49], bugs [12, 41, 42, 51] hardware faults [62], and the failure symptoms [33, 56], and many of these studies have made significant impact in that they led to tools capable of identifying many bugs (e.g., [16, 39]). However, the entire manifestation sequence connecting them is far less well-understood.

For each failure considered, we carefully studied the failure report, the discussion between users and developers, the logs and the code, and we manually reproduced 73 of the failures to better understand the specific manifestations that occurred.

Overall, we found that the error manifestation sequences tend to be relatively complex: more often than not, they require an unusual sequence of multiple events with specific input parameters from a large space to lead the system to a failure. This is perhaps not surprising considering that these systems have undergone thorough testing using unit tests, random error injections [18], and static bug finding tools such as FindBugs [32], and they are deployed widely and in constant use at many organizations. But it does suggest that top-down testing, say

1Throughout this paper, we use the following standard terminology [36]. A fault is the initial root cause, which could be a hardware malfunction, a software bug, or a misconfiguration. A fault can produce abnormal behaviors referred to as errors, such as system call error return or Java exceptions. Some of the errors will have no user-visible side-effects or may be appropriately handled by software; other errors manifest into a failure, where the system malfunction is noticed by end users or operators.
using input and error injection techniques, will be challenged by the large input and state space. This is perhaps why these studied failures escaped the rigorous testing used in these software projects.

We further studied the characteristics of a specific subset of failures — the catastrophic failures that affect all or a majority of users instead of only a subset of users. Catastrophic failures are of particular interest because they are the most costly ones for the vendors, and they are not supposed to occur as these distributed systems are designed to withstand and automatically recover from component failures. Specifically, we found that:

*almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software.*

While it is well-known that error handling code is often buggy [24, 44, 55], its sheer prevalence in the causes of the catastrophic failures is still surprising. Even more surprising given that the error handling code is the last line of defense against failures, we further found that:

*in 58% of the catastrophic failures, the underlying faults could easily have been detected through simple testing of error handling code.*

In fact, in 35% of the catastrophic failures, the faults in the error handling code fall into three trivial patterns: (i) the error handler is simply empty or only contains a log printing statement, (ii) the error handler aborts the cluster on an overly-general exception, and (iii) the error handler contains expressions like “FIXME” or “TODO” in the comments. These faults are easily detectable by tools or code reviews without a deep understanding of the runtime context. In another 23% of the catastrophic failures, the error handling logic of a non-fatal error was so wrong that any statement coverage testing or more careful code reviews by the developers would have caught the bugs.

To measure the applicability of the simple rules we extracted from the bugs that have lead to catastrophic failures, we implemented Aspirator, a simple static checker. Aspirator identified 121 new bugs and 379 bad practices in 9 widely used, production quality distributed systems, despite the fact that these systems already use state-of-the-art bug finding tools such as FindBugs [32] and error injection tools [18]. Of these, 143 have been fixed or confirmed by the systems’ developers.

Our study also includes a number of additional observations that may be helpful in improving testing and debugging strategies. We found that 74% of the failures are deterministic in that they are guaranteed to manifest with an appropriate input sequence, that almost all failures are guaranteed to manifest on no more than three nodes, and that 77% of the failures can be reproduced by a unit test.

Moreover, in 76% of the failures, the system emits explicit failure messages; and in 84% of the failures, all of the triggering events that caused the failure are printed into the log before failing. All these indicate that the failures can be diagnosed and reproduced in a reasonably straightforward manner, with the primary challenge being to have to sift through relatively noisy logs.

## 2 Methodology and Limitations

We studied 198 randomly sampled, real world failures reported on five popular distributed data-analytic and storage systems, including HDFS, a distributed file system [27]; Hadoop MapReduce, a distributed data-analytic framework [28]; HBase and Cassandra, two NoSQL distributed databases [2, 3]; and Redis, an in-memory key-value store supporting master/slave replication [54]. We focused on distributed data-intensive systems because they are the building blocks of many internet software services, and we selected the five systems because they are widely used and are considered production quality.

The failures we studied were extracted from the issue tracking databases of these systems. We selected tickets from these databases because of their high quality: each selected failure ticket documents a distinct failure that is confirmed by the developers, the discussions between users and developers, and the failure resolutions in the form of a patch or configuration change. Duplicate failures were marked by the developers, and are excluded from our study.

The specific set of failures we considered were selected from the issue tracking databases as follows. First, we only selected severe failures with the failure ticket *priority field* marked as “Blocker”, “Critical”, or “Major”. Secondly, we only considered tickets dated 2010 or later so as not to include failures of obsolete systems or systems early in their lifetime. Thirdly, we filtered out failures in testing systems by heuristically rejecting failures where the reporter and assignee (i.e., the developer who is assigned to resolve the failure) were the same. Finally, we randomly selected failures from the remaining set to make our observations representative of the entire failure population. Table 1 shows the distribution of the failures we studied, and the catastrophic ones from the sample set.

<table>
<thead>
<tr>
<th>Software</th>
<th>lang.</th>
<th>total failures</th>
<th>sampled failures</th>
<th>catastrophic failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>Java</td>
<td>3,923</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>HBase</td>
<td>Java</td>
<td>5,804</td>
<td>41</td>
<td>21</td>
</tr>
<tr>
<td>HDFS</td>
<td>Java</td>
<td>2,828</td>
<td>41</td>
<td>9</td>
</tr>
<tr>
<td>MapReduce</td>
<td>Java</td>
<td>3,469</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>Redis</td>
<td>C</td>
<td>1,192</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>17,216</strong></td>
<td><strong>198</strong></td>
<td><strong>48</strong></td>
</tr>
</tbody>
</table>

Table 1: Number of reported and sampled failures for the systems we studied, and the catastrophic ones from the sample set.
failure sets considered amongst the five systems and their sampling rates.

For each sampled failure ticket, we carefully studied the failure report, the discussion between users and developers, related error logs, the source code, and patches to understand the root cause and its propagation leading to the failure. We also manually reproduced 73 of the failures to better understand them.

Limitations: as with all characterization studies, there is an inherent risk that our findings may not be representative. In the following we list potential sources of biases and describe how we used our best-efforts to address them.

(1) Representativeness of the selected systems. We only studied distributed, data-intensive software systems. As a result, our findings might not generalize to other types of distributed systems such as telecommunication networks or scientific computing systems. However, we took care to select diverse types of data-intensive programs that include both data-storage and analytical systems, both persistent store and volatile caching, both written in Java and C, both master-slave and peer-to-peer designs. (HBase, HDFS, Hadoop MapReduce, and Redis use master-slave design, while Cassandra uses a peer-to-peer gossiping protocol.) At the very least, these projects are widely used: HDFS and Hadoop MapReduce are the main elements of the Hadoop platform, which is the predominant big-data analytic solution [29]; HBase and Cassandra are the top two most popular wide column store system [30], and Redis is the most popular key-value store system [53].

Our findings also may not generalize to systems earlier in their development cycle since we only studied systems considered production quality. However, while we only considered tickets dated 2010 or later to avoid bugs in premature systems, the buggy code may have been newly added. Studying the evolutions of these systems to establish the correlations between the bug and the code’s age remains as the future work.

(2) Representativeness of the selected failures. Another potential source of bias is the specific set of failures we selected. We only studied tickets found in the issue-tracking databases that are intended to document software bugs. Other errors, such as misconfigurations, are more likely to be reported in user discussion forums, which we chose not to study because they are much less rigorously documented, lack authoritative judgements, and are often the results of trivial mistakes. Consequently, we do not draw any conclusions on the distribution of faults, which has been well-studied in complementary studies [50, 52]. Note, however, that it can be hard for a user to correctly identify the nature of the cause of a failure; therefore, our study still includes failures that stem from misconfigurations and hardware faults.

In addition, we excluded duplicated bugs from our study so that our study reflects the characteristics of distinct bugs. One could argue that duplicated bugs should not be removed because they happened more often. There were only a total of 10 duplicated bugs that were excluded from our original sample set. Therefore they would not significantly change our conclusions even if they were included.

(3) Size of our sample set. Modern statistics suggests that a random sample set of size 30 or more is large enough to represent the entire population [57]. More rigorously, under standard assumptions, the Central Limit Theorem predicts a 6.9% margin of error at the 95% confidence level for our 198 random samples. Obviously, one can study more samples to further reduce the margin of error.

(4) Possible observer errors. To minimize the possibility of observer errors in the qualitative aspects of our study, all inspectors used the same detailed written classification methodology, and all failures were separately investigated by two inspectors before consensus was reached.

3 General Findings

This section discusses general findings from the entire failure data set in order to provide a better understanding as to how failures manifest themselves. Table 2 categorizes the symptoms of the failures we studied.

Overall, our findings indicate that the failures are relatively complex, but they identify a number of opportunities for improved testing. We also show that the logs produced by these systems are rich with information, making the diagnosis of the failures mostly straightforward. Finally, we show that the failures can be reproduced offline relatively easily, even though they typically occurred on long-running, large production clusters. Specifically, we show that most failures require no more 3 nodes and no more than 3 input events to reproduce, and most failures are deterministic. In fact, most of them can be reproduced with unit tests.

<table>
<thead>
<tr>
<th>Symptom</th>
<th>all</th>
<th>catastrophic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unexpected termination</td>
<td>74</td>
<td>17 (23%)</td>
</tr>
<tr>
<td>Incorrect result</td>
<td>44</td>
<td>1 (2%)</td>
</tr>
<tr>
<td>Data loss or potential data loss*</td>
<td>40</td>
<td>19 (48%)</td>
</tr>
<tr>
<td>Hung System</td>
<td>23</td>
<td>9 (39%)</td>
</tr>
<tr>
<td>Severe performance degradation</td>
<td>12</td>
<td>2 (17%)</td>
</tr>
<tr>
<td>Resource leak/exhaustion</td>
<td>5</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Total</td>
<td>198</td>
<td>48 (24%)</td>
</tr>
</tbody>
</table>

Table 2: Symptoms of failures observed by end-users or operators. The right-most column shows the number of catastrophic failures with “%” identifying the percentage of catastrophic failures over all failures with a given symptom. *: examples of potential data loss include under-replicated data blocks.
which is why only in minority of cases will a single input event actually trigger a failure, since the event often involves rarely used or newly introduced features, or are caused by concurrency bugs. Most failures require multiple preceding events, so the sum of the “%” column is greater than 100%.

### 3.1 Complexity of Failures

Overall, our findings indicate that the manifestations of the failures are relatively complex.

**Finding 1** A majority (77%) of the failures require more than one input event to manifest, but most of the failures (90%) require no more than 3. (See Table 3.)

Figure 1 provides an example where three input events are required for the failure to manifest.

Table 4 categorizes the input events that lead to failures. The % column reports the percentage of failure where the input event is required to trigger the failure. Most failures require multiple preceding events, so the sum of the “%” column is greater than 100%.

**Table 3:** Minimum number of input events required to trigger the failures.

<table>
<thead>
<tr>
<th>Num. of events</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23%</td>
</tr>
<tr>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>17%</td>
</tr>
<tr>
<td>4</td>
<td>5%</td>
</tr>
<tr>
<td>&gt; 4</td>
<td>5%</td>
</tr>
</tbody>
</table>

Table 4: Input events that led to failures. The % column reports the percentage of failure where the input event is required to trigger the failure. Most failures require multiple preceding events, so the sum of the “%” column is greater than 100%.

<table>
<thead>
<tr>
<th>Input event type</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting a service</td>
<td>58%</td>
</tr>
<tr>
<td>File/database write from client</td>
<td>32%</td>
</tr>
<tr>
<td>Unreachable node (network error, crash, etc.)</td>
<td>24%</td>
</tr>
<tr>
<td>Configuration change</td>
<td>23%</td>
</tr>
<tr>
<td>Adding a node to the running system</td>
<td>15%</td>
</tr>
<tr>
<td>File/database read from client</td>
<td>13%</td>
</tr>
<tr>
<td>Node restart (intentional)</td>
<td>9%</td>
</tr>
<tr>
<td>Data corruption</td>
<td>3%</td>
</tr>
<tr>
<td>Other</td>
<td>4%</td>
</tr>
</tbody>
</table>

**Finding 2** The specific order of events is important in 88% of the failures that require multiple input events.

Obviously, most of the individual events in Table 4 are heavily exercised and tested (e.g., read and write), which is why only in minority of cases will a single input event induce a failure. In most cases, a specific combination and sequence of multiple events is needed to transition the system into a failed state. Consider the failure example shown in Figure 1. While the events “upload file”, “append to file”, and “add another datanode” are not problematic individually, the combination of the first two will lead the system into an error state, and the last event actually triggers the failure.

Finding 1 and 2 show the complexity of failures in large distributed system. To expose the failures in testing, we need to not only explore the combination of multiple input events from an exceedingly large event space, we also need to explore different permutations.

### 3.2 Opportunities for Improved Testing

Additional opportunities to improve existing testing strategies may be found when considering the types of input events required for a failure to manifest. We briefly discuss some of the input event types of Table 4.

**Starting up services:** More than half of the failures require the start of some services. This suggests that the starting of services — especially more obscure ones — should be more heavily tested. About a quarter of the failures triggered by starting a service occurred on systems that have been running for a long time; e.g., the HBase “Region Split” service is started only when a table grows larger than a threshold. While such a failure may seem hard to test since it requires a long running system, it can be exposed intentionally by forcing a start of the service during testing.
Unreachable nodes: 24% of the failures occur because a node is unreachable. This is somewhat surprising given that network errors and individual node crashes are expected to occur regularly in large data centers [14]. This suggests that tools capable of injecting network errors systematically [18, 23, 65] should be used more extensively when inputting other events during testing.

Configuration changes: 23% of the failures are caused by configuration changes. Of those, 30% involve misconfigurations. The remaining majority involve valid changes to enable certain features that may be rarely-used. While the importance of misconfigurations have been observed in previous studies [22, 50, 66], only a few techniques exist to automatically explore configuration changes and test the resulting reaction of the system [19, 40, 63]. This suggests that testing tools should be extended to combine (both valid and invalid) configuration changes with other operations.

Adding a node: 15% of the failures are triggered by adding a node to a running system. Figure 1 provides an example. This is somewhat alarming, given that elastically adding and removing nodes is one of the principle promises of “cloud computing”. It suggests that adding nodes needs to be tested under more scenarios.

The production failures we studied typically manifested themselves on configurations with a large number of nodes. This raises the question of how many nodes are required for an effective testing and debugging system.

Finding 3 Almost all (98%) of the failures are guaranteed to manifest on no more than 3 nodes. 84% will manifest on no more than 2 nodes. (See Table 5.)

The number is similar for catastrophic failures. Finding 3 implies that it is not necessary to have a large cluster to test for and reproduce failures.

Note that Finding 3 does not contradict the conventional wisdom that distributed system failures are more likely to manifest on large clusters. In the end, testing is a probabilistic exercise. A large cluster usually involves more diverse workloads and fault modes, thus increasing the chances for failures to manifest. However, what our finding suggests is that it is not necessary to have a large cluster of machines to expose bugs, as long as the specific sequence of input events occurs.

We only encountered one failure that required a larger number of nodes (over 1024): when the number of simultaneous Redis client connections exceeded the OS limit, epoll() returned error, which was not handled properly, causing the entire cluster to hang. All of the other failures require fewer than 10 nodes to manifest.

3.3 The Role of Timing

A key question for testing and diagnosis is whether the failures are guaranteed to manifest if the required sequence of input events occur (i.e., deterministic failures), or not (i.e., non-deterministic failures)?

Finding 4 74% of the failures are deterministic — they are guaranteed to manifest given the right input event sequences. (See Table 6.)

This means that for a majority of the failures, we only need to explore the combination and permutation of input events, but no additional timing relationship. This is particularly meaningful for testing those failures that require long-running systems to manifest. As long as we can simulate those events which typically only occur on long running systems (e.g., region split in HBase typically only occurs when the region size grows too large), we can expose these deterministic failures. Moreover, the failures can still be reproduced after inserting additional log output, enabling tracing, or using debuggers.

Finding 5 Among the 51 non-deterministic failures, 53% have timing constraints only on the input events. (See Table 7.)

These constraints require an input event to occur either before or after some software internal execution event such as a procedure call. Figure 2 shows an example. In addition to the order of the four input events (that can be

### Table 5: Min. number of nodes needed to trigger the failures.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>cumulative distribution function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>all failures 37%</td>
</tr>
<tr>
<td>2</td>
<td>catastrophic 43%</td>
</tr>
<tr>
<td>3</td>
<td>total 84%</td>
</tr>
<tr>
<td>&gt; 3</td>
<td>98%</td>
</tr>
<tr>
<td>&gt; 3</td>
<td>100%</td>
</tr>
</tbody>
</table>

### Table 6: Number of failures that are deterministic.

<table>
<thead>
<tr>
<th>Software</th>
<th>num. of deterministic failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>76% (31/41)</td>
</tr>
<tr>
<td>HBase</td>
<td>71% (29/41)</td>
</tr>
<tr>
<td>HDFS</td>
<td>76% (31/41)</td>
</tr>
<tr>
<td>MapReduce</td>
<td>63% (24/38)</td>
</tr>
<tr>
<td>Redis</td>
<td>79% (30/38)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>74% (147/198)</td>
</tr>
</tbody>
</table>

### Table 7: Break-down of the non-deterministic failures. The “other” category is caused by nondeterministic behaviors from the OS and third party libraries.

<table>
<thead>
<tr>
<th>Source of non-determinism</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing btw. input event &amp; internal exe. event</td>
<td>27 (53%)</td>
</tr>
<tr>
<td>Multi-thread atomicity violation</td>
<td>13 (25%)</td>
</tr>
<tr>
<td>Multi-thread deadlock</td>
<td>3 (6%)</td>
</tr>
<tr>
<td>Multi-thread lock contention (performance)</td>
<td>4 (8%)</td>
</tr>
<tr>
<td>Other</td>
<td>4 (8%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>51 (100%)</td>
</tr>
</tbody>
</table>
HBase MapReduce.

Redis HDFS

HLog that does not contain the newly written data. HMaster assigns a new region server, it only recovered the old (boxed). Some newly written data will be lost because when HMaster assigns a new region server, it only recovered the old HLog that does not contain the newly written data.

![Diagram](image)

Figure 2: A non-deterministic failure in HBase with timing requirements (shown with solid arrows) only on input events (boxed). Some newly written data will be lost because when HMaster assigns a new region server, it only recovered the old HLog that does not contain the newly written data.

```plaintext
write_lock();
/* remove a
* Large directory */
write_unlock();
```

Critical region is too large, causing concurrent write requests to hang.

Figure 3: Performance degradation in HDFS caused by a single request to remove a large directory.

controlled by a tester), the additional requirement is that the client write operations must occur before HMaster assigns the region to a new Region Server, which cannot be completely controlled by the user.

Note that these non-deterministic dependencies are still easier to test and debug than non-determinisms stemming from multi-threaded interleavings, since at least one part of the timing dependency can be controlled by testers. Testers can carefully control the timing of the input events to induce the failure. Unit tests and model checking tools can further completely manipulate such timing dependencies by controlling the timing of both the input events and the call of internal procedures. For example, as part of the patch to fix the bug in Figure 2, developers used a unit test that simulated the user inputs and the dependencies with HMaster’s operations to deterministically reproduce the failure.

The majority of the remaining 24 non-deterministic failures stem from shared-memory multi-threaded interleavings. We observed three categories of concurrency bugs in our dataset: atomicity violation [42], deadlock, and lock contention that results in performance degradation. It is much harder to expose and reproduce such failures because it is hard for users or tools to control timing, and adding a single logging statement can cause the failure to no longer expose itself. We reproduced 10 of these non-deterministic failures and found the atomicity violations and deadlocks the most difficult to reproduce (we had to manually introduce additional timing delays, like `Thread.sleep()` in the code to trigger the bugs). The lock contention cases, however, are not as difficult to reproduce. Figure 3 shows an example where a bug caused unnecessary lock contention.

### 3.4 Logs Enable Diagnosis Opportunities

Overall, we found the logs output by the systems we studied to be rich with useful information. We assume the default logging verbosity level is used.

**Finding 6** 76% of the failures print explicit failure-related error messages. (See Figure 4.)

This finding somewhat contradicts the findings of our previous study [67] on failures in non-distributed systems, including Apache httpd, PostgreSQL, SVN, squid, and GNU Coreutils, where only 43% of failures had explicit failure-related error messages logged. We surmise there are three possible reasons why developers output log messages more extensively for the distributed systems we studied. First, since distributed systems are more complex, and harder to debug, developers likely pay more attention to logging. Second, the horizontal scalability of these systems makes the performance overhead of outputing log message less critical. Third, communicating through message-passing provides natural points to log messages; for example, if two nodes cannot communicate with each other because of a network problem, both have the opportunity to log the error.

**Finding 7** For a majority (84%) of the failures, all of their triggering events are logged. (See Figure 4.)

This suggests that it is possible to deterministically replay the majority of failures based on the existing log messages alone. Deterministic replay has been widely explored by the research community [4, 13, 15, 26, 35, 47, 61]. However, these approaches are based on intrusive tracing with significant runtime overhead and the need to modify software/hardware.

**Finding 8** Logs are noisy: the median of the number of log messages printed by each failure is 824.
input event messages by filtering out the irrelevant ones. Existing log analysis techniques [5, 6, 48, 64] amine almost every log message. It would be helpful if those that triggered the failure are often logged at INFO level. However, the input events will reduce noise since a vast majority of the printed log messages are at INFO level. Therefore to further infer the input events one has to examine almost every log message. It would be helpful if existing log analysis techniques [5, 6, 48, 64] and tools were extended so they can infer the relevant error and input event messages by filtering out the irrelevant ones.

3.5 Failure Reproducibility

Conventional wisdom has it that failures which occur on large, distributed system in production are extremely hard to reproduce off-line. The users’ input may be unavailable due to privacy concerns, the difficulty in setting up an environment that mirrors the one in production, and the cost of third-party libraries, are often reasons cited as to why it is difficult for vendors to reproduce production failures. Our finding below indicates that failure reproduction might not be as hard as it is thought to be.

Finding 9 A majority of the production failures (77%) can be reproduced by a unit test. (See Table 8.)

While this finding might sound counter-intuitive, it is not surprising given our previous findings because: (1) in Finding 4 we show that 74% of the failures are deterministic, which means the failures can be reproduced with the same operation sequence; and (2) among the remaining non-deterministic failures, in 53% of the cases the timing can be controlled through unit tests.

Specific data values are not typically required to reproduce the failures; in fact, none of the studied failures required specific values of user’s data contents. Instead, only the required input sequences (e.g., file write, disconnect a node, etc.) are needed.

Figure 6 shows how a unit test can simulate the non-deterministic failure of Figure 2. It simulates a mini-cluster by starting three processes running as three nodes. It further simulates the key input events, including HMaster’s log split, Region Server’s log rolling, and the write requests. The required dependency where the client must send write requests before the master re-assigns the recovered region is also controlled by this unit test.

The failures that cannot be reproduced easily either depend on a particular execution environment (such as OS version or third party libraries), or were caused by non-deterministic thread interleavings.

4 Catastrophic Failures

Table 2 in Section 3 shows that 48 failures in our entire failure set have catastrophic consequences. We classify a failure to be catastrophic when it prevents all or a majority of the users from their normal access to the system. In practice, these failures result in cluster-wide outage, a hung cluster, or a loss to all or a majority of the user data. Note that a bug resulting in under-replicated data blocks is not considered as catastrophic, even when it affect all data blocks, because it does not prevent users from their normal read and write to their data yet. We specifically study the catastrophic failures because they are the ones with the largest business impact to the vendors.

The fact that there are so many catastrophic failures is perhaps surprising given that the systems considered all have High Availability (HA) mechanisms designed to prevent component failures from taking down the entire service. For example, all of the four systems with a master-slave design — namely HBase, HDFS, MapReduce, and Redis — are designed to, on a master node fail, automatically elect a new master node and fail.
Cassandra is a peer-to-peer system, thus by design it avoids single points of failure. Then why do catastrophic failures still occur?

**Finding 10** Almost all catastrophic failures (92%) are the result of incorrect handling of non-fatal errors explicitly signaled in software. (See Figure 5.)

These catastrophic failures are the result of more than one fault triggering, where the initial fault, whether due to a hardware fault, a misconfiguration, or a bug, first manifests itself explicitly as a non-fatal error — for example by throwing an exception or having a system call return an error. This error need not be catastrophic; however in the vast majority of cases, the handling of the explicit error was faulty, resulting in an error manifesting itself as a catastrophic failure. This prevalence of incorrect error handling is unique to catastrophic failures. In comparison, only 25% of the non-catastrophic failures in our study involve incorrect error handling, indicating that in non-catastrophic failures, error handling was mostly effective in preventing the errors from taking down the entire service.

Overall, we found that the developers are good at anticipating possible errors. In all but one case, the errors were checked by the developers. The only case where developers did not check the error was an unchecked error system call return in Redis. This is different from the characteristics observed in previous studies on file system bugs [24, 41, 55], where many errors weren’t even checked. This difference is likely because (i) the Java compiler forces developers to catch all the checked exceptions; and (ii) a variety of errors are expected to occur in large distributed systems, and the developers program more defensively. However, we found they were often simply sloppy in handling these errors. This is further corroborated in Findings 11 and 12 below. To be fair, we should point out that our findings are skewed in the sense that our study did not expose the many errors that are correctly caught and handled.

Nevertheless, the correctness of error handling code is particularly important given their impact. Previous studies [50, 52] show that the initial faults in distributed system failures are highly diversified (e.g., bugs, misconfigurations, node crashes, hardware faults), and in practice it is simply impossible to eliminate them all in large data centers [14]. It is therefore unavoidable that some of these faults will manifest themselves into errors, and error handling then becomes the last line of defense [45].

Of the catastrophic failures we studied, only four were not triggered by incorrect error handling. Three of them were because the servers mistakenly threw fatal exceptions that terminated all the clients, i.e., the clients’ error handling was correct. The other one was a massive performance degradation when a bug disabled DNS look-up result caching.

### 4.1 Trivial Mistakes in Error Handlers

**Finding 11** 35% of the catastrophic failures are caused by trivial mistakes in error handling logic — ones that simply violate best programming practices; and that can be detected without system specific knowledge.

Figure 5 further breaks down the mistakes into three categories: (i) the error handler ignores explicit errors; (ii) the error handler over-catches an exception and aborts the system; and (iii) the error handler contains “TODO” or “FIXME” in the comment.

25% of the catastrophic failures were caused by ignoring explicit errors (an error handler that only logs the error is also considered as ignoring the error). For systems written in Java, the exceptions were all explicitly thrown, whereas in Redis they were system call error returns. Figure 7 shows a data loss in HBase caused by ignoring an exception. Ignoring errors and allowing them to propagate is known to be bad programming practice [7, 60], yet we observed this lead to many catastrophic failures. At least the developers were careful at logging the errors: all the errors were logged except for one case where the Redis developers did not log the error system call return.

---

2 We assume the HA feature is always enabled when classifying catastrophic failures. We did not classify failures as catastrophic if HA was not enabled and the master node failed, even though it would likely have affected all the users of the system. This is because such failures are not unique compared to the other failures we studied — they just happened to have occurred on the master node.
Consequently, when a glitch in the namenode caused the developers catch a high-level exception: Throwable. inted only for IncorrectVersionException. However, exit() tions. Figure 8 shows such an example. The exit() was intended only for IncorrectVersionException. However, the developers catch a high-level exception: Throwable. Consequently, when a glitch in the namenode caused registerDatanode() to throw RemoteException, it was over-caught by Throwable and thus brought down every datanode. The fix was to handle RemoteException explicitly, so that only IncorrectVersionException would fall through. However, this is still bad practice since later when the code evolves, some other exceptions may be over-caught again. The safe practice is to catch the precise exception [7].

Figure 9 shows an even more obvious mistake, where the developers only left a comment “TODO” in the handler logic in addition to a logging statement. While this error would only occur rarely, it took down a production cluster of 4,000 nodes.

4.2 System-specific Bugs

The other 57% of the catastrophic failures are caused by incorrect error handling where system-specific knowledge is required to detect the bugs. (See Figure 5.)

Finding 12 In 23% of the catastrophic failures, while the mistakes in error handling were system specific, they are still easily detectable. More formally, the incorrect error handling in these cases would be exposed by 100% statement coverage testing on the error handling logic.

In other words, once the problematic basic block in the error handling code is triggered, the failure is guaranteed to be exposed. This suggests that these basic blocks were completely faulty and simply never properly tested. Figure 10 shows such an example. Once a test case can deterministically trigger KeeperException, the catastrophic failure will be triggered with 100% certainty.

Hence, a good strategy to prevent these failures is to start from existing error handling logic and try to reverse engineer test cases that trigger them. For example, symbolic execution techniques [8, 10] could be extended to purposefully reconstruct an execution path that can reach the error handling code block, instead of blindly exploring every execution path from the system entry points.

While high statement coverage on error handling code might seem difficult to achieve, aiming for higher statement coverage in testing might still be a better strategy than a strategy of applying random fault injections. For example, the failure in Figure 10 requires a very rare combination of events to trigger the buggy error handler. Our finding suggests that a “bottom-up” approach could be more effective: start from the error handling logic and reverse engineer a test case to expose errors there.

Existing testing techniques for error handling logic primarily use a “top-down” approach: start the system using testing inputs or model-checking [23, 65], and actively inject errors at different stages [9, 18, 44]. Tools like LFI [44] and FateDestini [23] are intelligent to inject errors only at appropriate points and avoid duplicated injections. Such techniques inevitably have greatly
improved the reliability of software systems. In fact, Hadoop developers have their own error injection framework to test their systems [18], and the production failures we studied are likely the ones missed by such tools.

However, our findings suggest that it could be challenging for such “top-down” approaches to further expose these remaining production failures. They require rare sequence of input events to first take the system to a rare state, before the injected error can take down the service. In addition, 38% of the failures only occur in long-running systems. Therefore, the possible space of input events would simply be untractable.

**Complex bugs:** the remaining 34% of catastrophic failures involve complex bugs in the error handling logic. These are the cases where developers did not anticipate certain error scenarios. As an example, consider the failure shown in Figure 11. While the handling logic makes sense for a majority of the checksum errors, it did not consider the scenario where a single client reports a massive number of corruptions (due to corrupt RAM) in a very short amount of time. These type of errors — which are almost byzantine — are indeed the hardest to test for. Detecting them require both understanding how the system works and anticipating all possible real-world failure modes. While our study cannot provide constructive suggestions on how to identify such bugs, we found they only account for one third of the catastrophic failures.

4.3 Discussion

While we show that almost all of the catastrophic failures are the result of incorrect error handling, it could be argued that most of the code is reachable from error handling blocks in real-world distributed systems, therefore most of the bugs are “incorrect error handling”. However, our findings suggest many of the bugs can be detected by only examining the exception handler blocks (e.g., the `catch` block in Java). As we show will in Table 9, the number of `catch` blocks in these systems is relatively small. For example, in HDFS, there are only 2652 `catch` blocks. In particular, the bugs belonging to the “trivial mistakes” category in Finding 11 can be easily detected by only examining these `catch` blocks.

An interesting question is whether the outages from large internet software vendors are also the result of incorrect error handling. While we cannot answer this rigorously without access to their internal failure databases, the postmortem analysis of some of the most visible outages are released to the public. Interestingly, some of the anecdotal outages are the result of incorrect error handling. For example, in an outage that brought down facebook.com for approximately 2.5 hours, which at that time was “the worst outage Facebook have had in over four years”, “the key flaw that caused the outage to be so severe was an unfortunate handling of an error condition” [17]. In the outage of Amazon Web Services in 2011 [59] that brought down Reddit, Quora, FourSquare, parts of the New York Times website, and about 70 other sites, the initial cause was a configuration change that mistakenly routed production traffic to a secondary network that was not intended for a heavy workload. Consequently, nodes start to fail. What lead this to further propagate into a service-level failure was the incorrect handling of node-failures — “the nodes failing to find new nodes did not back off aggressively enough when they could not find space, but instead, continued to search repeatedly”. This caused even more network traffic, and eventually lead to the service-level failure.

5 Aspirator: A Simple Checker

In Section 4.1, we observed that some of the most catastrophic failures are caused by trivial mistakes that fall into three simple categories: (i) error handler is empty; (ii) error handler over-catches exceptions and aborts; and (iii) error handler contains phrases like “TODO” and “FIXME”. To measure the applicability of these simple rules, we built a rule-based static checker, Aspirator, capable of locating these bug patterns. Next we discuss how Aspirator is implemented and the results of applying it to a number of systems.

5.1 Implementation of Aspirator

We implemented Aspirator using the Chord static analysis framework [11] on Java bytecode. Aspirator works as follows: it scans Java bytecode, instruction by instruction. If an instruction can throw exception `e`, Aspirator identifies and records the corresponding `catch` block for `e`. Aspirator emits a warning if the `catch` block is empty or just contains a log printing statement, or if the `catch` block contains “TODO” or “FIXME” com-
ments in the corresponding source code. It also emits
a warning if a catch block for a higher-level exception (e.g., Exception or Throwable) might catch multiple
lower-level exceptions and at the same time calls abort
or System.exit(). Aspirator is capable of identifying
these over-catches because when it reaches a catch
block, it knows exactly which exceptions from which instructions the catch block handles.

Not every empty catch block is necessarily a bad
practice or bug. Consider the following example where the exception is handled outside of the catch block:

```
uri = null;
try {
    uri = Util.fileAsURI(new File(uri));
} catch (IOException ex) { /* empty */ }
```

Therefore Aspirator will not emit a warning on an empty
catch block if both of the following conditions are true:
(i) the corresponding try block modifies a variable V;
and (ii) the value of V is checked in the basic block
following the catch block. In addition, if the last in-
struction in the corresponding try block is a return,
break, or continue, and the block after the catch
block is not empty, Aspirator will not report a warning
if the catch block is empty because all the logic after
the catch block is in effect exception handling.

Aspirator further provides runtime configuration options
to allow programmers to adjust the trade-offs between false positives and false negatives. It allows pro-
grammers to specify exceptions that should not result in
a warning. In our testing, we ignored all instances of the
FileNotFoundException exception, because we found the vast ma-
jority of them do not indicate a true error. Aspirator also
allows programmers to exclude certain methods from the
analysis. In our testing, we use this to suppress warnings
if the ignored exceptions are from a shutdown, close or
cleanup method — exceptions during a cleanup phase are likely less important because the system is being
brought down anyway. Using these two heuristics did not
affect Aspirator’s capability to detect the trivial mistakes
leading to catastrophic failures in our study, yet signifi-
cantly reduce the number of false positives.

Limitations: As a proof-of-concept, Aspirator currently
only works on Java and other languages that are compat-
ible with Java bytecode (e.g., Scala), where exceptions
are supported by the language and are required to be ex-
plitly caught. The main challenge to extend Aspirator
to non-Java programs is to identify the error conditions.
However, some likely error conditions can still be eas-
ily identified, including system call error returns, switch
fall-through, and calls to abort().

In addition, Aspirator cannot estimate the criticality of
the warnings it emits. Hence, not every warning emitted
will identify a bug that could lead to a failure; in fact,
some false positives are emitted. However, because As-
pirator provides, with each warning, a list of caught ex-
ceptions together with the instructions that throw them,
developers in most cases will be able to quickly assess the criticality of each warning and possibly annotate the
program to suppress specific future warnings.

Finally, the functionality of Aspirator could (and prob-
ably should) be added to existing static analysis tools,
such as FindBugs [32].

5.2 Checking Real-world Systems

We first evaluated Aspirator on the set of catastrophic
failures used in our study. If Aspirator had been used and
the captured bugs fixed, 33% of the Cassandra, HBase,
HDFS, and MapReduce’s catastrophic failures we stud-
ied could have been prevented.

We then used Aspirator to check the latest stable ver-
sions of 9 distributed systems or components used to
build distributed systems (e.g., Tomcat web-server). As-
pirator’s analysis finishes within 15 seconds for each sys-
tem on a MacBook Pro laptop with 2.7GHz Intel Core i7
processor, and has memory footprints of less than 1.2GB.

We categorize each warning generated by Aspirator
into one of three categories: bug, bad practice, and false
positive. For each warning, we use our best-effort to
understand the consequences of the exception handling
logic. Warnings are categorized as bugs only if we could
definitively conclude that, once the exception occurs, the
handling logic could lead to a failure. They were cate-
gorized as false positives if we clearly understood they
would not lead to a failure. All other cases are those
that we could not definitively infer the consequences of
the exception handling logic without domain knowl-
edge. Therefore we conservatively categorize them as
bad practices.

Table 9 shows the results. Overall, Aspirator detected
500 new bugs and bad practices along with 115 false
positives. Note that most of these systems already run
state-of-the-art static checkers like FindBugs [32], which
checks for over 400 rules, on every code check-in. Yet
Aspirator has found new bugs in all of them.

Bugs: many bugs detected by Aspirator could indeed
lead to catastrophic failures. For example, all 4 bugs
caught by the abort-in-over-catch checker could bring
down the cluster on an unexpected exception in a similar fashion as in Figure 8. All 4 of them have been fixed.

Some bugs can also cause the cluster to hang. Aspirator detected 5 bugs in HBase and Hive that have a pattern similar to the one depicted in Figure 12 (a). In this example, when `tablelock` cannot be released, HBase only outputs an error message and continues executing, which can deadlock all servers accessing the table. The developers fixed this bug by immediately cleaning up the states and aborting the problematic server [31].

Figure 12 (b) shows a bug that could lead to data loss. An IOException could be thrown when HDFS is recovering user data by replaying the updates from the Edit log. Ignoring it could cause a silent data loss.

**Bad practices**: the bad practice cases include potential bugs for which we could not definitively determine their consequences without domain expertise. For example, if deleting a temporary file throws an exception and is subsequently ignored, it may be inconsequential. However, it is nevertheless considered a bad practice because it may indicate a more serious problem in the file system.

Some of these cases could as well be false positives. While we cannot determine how many of them are false positives, we did report 87 of the cases that we initially classified as “bad practices” to developers. Among them, 58 were confirmed or fixed, but 17 were rejected. The 17 rejected ones were subsequently classified as “false positives” in Table 9.

**False positives**: 19% of the warnings reported by Aspirator are false positives. Most of them are due to that Aspirator does not perform inter-procedural analysis. Consider the following example, where an exception is handled by testing the return value of a method call:

```java
try {
    set_A();
    } catch (SomeException e) { /* empty */ }
    if (!A_is_not_set())) {/* handle it here! */
```

In addition to `FileNotFoundException` and exceptions from from `shutdown, close, and cleanup`, Aspirator should have been further configured to exclude the warnings on other exceptions. For example, many of the false positives are caused by empty handlers of Java’s reflection related exceptions, such as `NoSuchFieldException`. Once programmers realize an exception should have been excluded from Aspirator’s analysis, they can simply add this exception to Aspirator’s configuration file.

### 5.3 Experience

**Interaction with developers**: We reported 171 bugs and bad practices to the developers through the official bug tracking website. To this date, 143 have already been confirmed or fixed by the developers (73 of them have been fixed, and the other 70 have been confirmed but not fixed yet), 17 were rejected, and the others have not received any responses.

We received mixed feedback from developers. On the one hand, we received some positive comments like: “I really want to fix issues in this line, because I really want us to use exceptions properly and never ignore them”, “No one would have looked at this hidden feature; ignoring exceptions is bad precisely for this reason”, and “catching Throwable [i.e., exception over-catch] is bad, we should fix these”. On the other hand, we received negative comments like: “I fail to see the reason to handle every exception”.

There are a few reasons for developers’ obliviousness to the handling of errors. First, these ignored errors may not be regarded as critical enough to be handled properly. Often, it is only until the system suffers serious failures will the importance of the error handling be realized by developers. We hope to raise developers’ awareness by showing that many of the most catastrophic failures today are caused precisely by such obliviousness to the correctness of error handling logic.

Secondly, the developers may believe the errors would never (or only very rarely) occur. Consider the following code snippet detected by Aspirator from HBase:
In this case, the developers thought the constructor could never throw an exception, so they ignored it (as per the comment in the code). We observed many empty error handlers contained similar comments in multiple systems we checked. We argue that errors that “can never happen” should be handled defensively to prevent them from propagating. This is because developers’ judgement could be wrong, later code evolutions may enable the error, and allowing such unexpected errors to propagate can be deadly. In the HBase example above, developers’ judgement was indeed wrong. The constructor is implemented as follows:

```java
public TimeRange (long min, long max)
throws IOException {
if (max < min)
throw new IOException("max < min");
}
```

It could have thrown an IOException when there is an integer overflow, and swallowing this exception could have lead to a data loss. The developers later fixed this by handling the IOException properly.

Thirdly, proper handling of the errors can be difficult. It is often much harder to reason about the correctness of a system’s abnormal execution path than its normal execution path. The problem is further exacerbated by the reality that many of the exceptions are thrown by third party components lacking of proper documentations. We surmise that in many cases, even the developers may not fully understand the possible causes or the potential consequences of an exception. This is evidenced by the following code snippet from CloudStack:

```java
try {
    t = new TimeRange(timestamp, timestamp+1);
} catch (IOException e) {
    // Will never happen
}
```

We observed similar comments from empty exception handlers in other systems as well.

Finally, in reality feature development is often prioritized over exception handling when release deadlines loom. We embarrassingly experienced this ourselves when we ran Aspirator on Aspirator’s code: we found 5 empty exception handlers, all of them for the purpose of catching exceptions thrown by the underlying libraries and put there only so that the code would compile.

**Good practice in Cassandra:** among the 9 systems we checked, Cassandra has the lowest bug-to-handler-block ratio, indicating that Cassandra developers are careful in following good programming practices in exception handling. In particular, the vast majority of the exceptions are handled by recursively propagating them to the callers, and are handled by top level methods in the call graphs. Interestingly, among the 5 systems we studied, Cassandra also has the lowest rate of catastrophic failures in its randomly sampled failure set (see Table 1).

### 6 Related Work

A number of studies have characterized failures in distributed systems, which led to a much deeper understanding of these failures and hence improved reliability. Our study is the first (to the best of our knowledge) analysis to understand the end-to-end manifestation sequence of these failures. The manual analysis allowed us to find the weakest link on the manifestation sequence for the most catastrophic failures, namely the incorrect error handling. While it is well-known that error handling is a source of many errors, we found that these bugs in error handling code, many of them extremely simple, are the dominant cause of today’s catastrophic failures.

Next, we discuss three categories of related work: characterization studies, studies on error handling code, and distributed system testing.

**Failure characterization studies** Oppenheimer et al. eleven years ago studied over 100 failure reports from deployed internet services [50]. They discussed the root causes, time-to-repair, and mitigation strategies of these failures, and summarized a series of interesting findings (e.g., operator mistakes being the most dominant cause). Our study is largely complementary since the open-source projects allow us to examine a richer source of data, including source code, logs, developers’ discussions, etc., which were not available for their study. Indeed, as acknowledged by the authors, they “could have been able to learn more about the detailed causes if [they] had been able to examine the system logs and bug tracking database”.

Rabkin and Katz [52] analyzed reports from Cloudera’s production hadoop clusters. Their study focused on categorizing the root causes of the failures.

Li et al. [38] studied bugs in Microsoft Bing’s data analytic jobs written in SCOPE. They found that most of the bugs were in the data processing logic and were often caused by frequent change of table schema.

Others studied bugs in non-distributed systems. In 1985, Gray examined over 100 failures from the Tandem [22] operating system, and found operator mistakes and software bugs to be the two major causes. Chou et al. [12] studied OS bugs and observed that device drivers are the most buggy. This finding led to many systems and tools to improve device driver quality, and a study [51] ten years later suggested that the quality of device drivers have indeed greatly improved. Lu et al. [42] studied concurrency bugs in server programs, and found many inter-
esting findings, e.g., almost all of the concurrency bugs can be triggered using 2 threads.

**Study on error handling code** Many studies have shown that error handling code is often buggy [24, 44, 55, 58]. Using a static checker, Gunawi *et al.* found that file systems and storage device drivers often do not correctly propagate error code [24]. Fu and Ryder also observed that a significant number of `catch` blocks were empty in many Java programs [20]. But they did not study whether they have caused failures. In a study on field failures with IBM’s MVS operating system between 1986 and 1989, Sullivan *et al.* found that incorrect error recovery was the cause of 21% of the failures and 36% of the failures with high impact [58]. In comparison, we find that in the distributed systems we studied, incorrect error handling resulted in 25% of the non-catastrophic failures, and 92% of the catastrophic ones.

Many testing tools can effectively expose incorrect error handling through error injections [18, 23, 44]. Fate&Destini [23] can intelligently inject unique combinations of multiple errors; LFI [44] selectively injects errors at the program/library boundary and avoids duplicate error injections. While these tools can be effective in exposing many incorrect error handling bugs, they all use a “top-down” approach and rely on users/testers to provide workloads to drive the system. In our study, we found that a combination of input events is needed to drive the system to the error state which is hard to trigger using a top-down approach. Our findings suggest that a “bottom-up” approach, which reconstruct test cases from the error handling logic, can effectively expose most faults that lead to catastrophic failures.

Other tools are capable of identify bugs in error handling code via static analysis [24, 55, 67]. EIO [24] uses static analysis to detect error code that is either unchecked or not further propagated. Errlog [67] reports error handling code that is not logged. In comparison, our simple checker is complementary. It detects exceptions that are checked but incorrectly handled, regardless whether they are logged or not.

**Distributed system testing** Model checking [25, 34, 37, 65] tools can be used to systematically explore a large combination of different events. For example, SAMC [37] can intelligently inject multiple errors to drive the target system into a corner case. Our study further helps users make informed decisions when using these tools (e.g., users need to check no more than three nodes).

### 7 Conclusions

This paper presented an in-depth analysis of 198 user-reported failures in five widely used, data-intensive distributed systems in the form of 12 findings. We found that the error manifestation sequences leading to the failures to be relatively complex. However, we also found that for the most catastrophic failures, almost all of them are caused by incorrect error handling, and 58% of them are trivial mistakes or can be exposed by statement coverage testing.

It is doubtful that existing testing techniques will be successful uncovering many of these error handling bugs. They all use a “top-down” approach: start the system using generic inputs or model-checking [65, 23], and actively inject errors at different stages [9, 18, 44]. However the size of the input and state space, and the fact that a significant number of failures only occur on long-running systems, makes the problem of exposing these bugs intractable. For example, Hadoop has its own error injection framework to test their system [18], but the production failures we studied are likely the ones missed by such tools.

Instead, we suggest a three pronged approach to expose these bugs: (1) use a tool similar to the Aspirator that is capable of identifying a number of trivial bugs; (2) enforce code reviews on error-handling code, since the error handling logic is often simply wrong; and (3) use, for example, extended symbolic execution techniques [8, 10] to purposefully reconstruct execution paths that can reach each error handling code block. Our detailed analysis of the failures and the source code of Aspirator are publicly available at: [http://www.eecg.toronto.edu/failureAnalysis/](http://www.eecg.toronto.edu/failureAnalysis/).

### Acknowledgements

We greatly appreciate the anonymous reviewers, our shepherd Jason Flinn, and Leonid Ryzhyk for their insightful feedback. This research is supported by NSERC Discovery grant, NetApp Faculty Fellowship, and Connaught New Researcher Award.

### References

of The International Conference on Software Engineering, ICSE’13, 2013.


