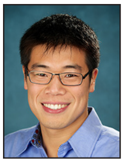
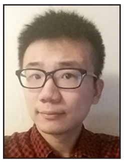
Simple Testing Can Prevent Most Critical Failures
An Analysis of Production Failures in Distributed Data-Intensive Systems

DING YUAN, YU LUO, XIN ZHUANG, GUILHERME RENNA RODRIGUES, XU ZHAO, YONGLE ZHANG, PRANAY U. JAIN, AND MICHAEL STUMM



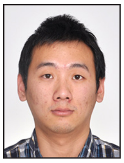
Ding Yuan is an assistant professor in the Electrical and Computer Engineering Department of the University of Toronto. He works in computer systems, with a focus on their reliability and performance. yuan@ece.toronto.edu

Yu (Jack) Luo is an undergraduate student at the University of Toronto studying computer engineering. He has interned at IBM working with memory management and disaster recovery. His research interests are in systems, failure recovery, and log analysis. jack.luo@mail.utoronto.ca



Xin Zhuang is an undergraduate at the University of Toronto studying computer engineering. His research interest is in software systems. xin.zhuang@mail.utoronto.ca

Guilherme Renna Rodrigues is an exchange student at the University of Toronto and is an undergraduate at CEFET-MG, Belo Horizonte, Brazil. The research project at the University of Toronto has led to his interest in practical solutions for computing systems problems. guilhermerenna@gmail.com



Xu Zhao is a graduate student at the University of Toronto. He received a B.Eng. in computer science from Tsinghua University, China. At U of T, his research is focused on reliability and performance of distributed systems. nuk.zhao@mail.utoronto.ca

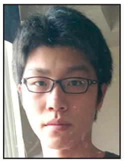
Large, production-quality distributed systems still fail periodically, sometimes catastrophically where most or all users experience an outage or data loss. Conventional wisdom has it that these failures can only manifest themselves on large production clusters and are extremely difficult to prevent a priori, because these systems are designed to be fault tolerant and are well-tested. By investigating 198 user-reported failures that occurred on production-quality distributed systems, we found that almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors, and, surprisingly, many of them are caused by trivial mistakes such as error handlers that are empty or that contain expressions like "FIXME" or "TODO" in the comments. We therefore developed a simple static checker, Aspirator, capable of locating trivial bugs in error handlers; it found 143 new bugs and bad practices that have been fixed or confirmed by the developers.

Our study also includes a number of additional observations that may be helpful in improving testing and debugging strategies. We found that from a testing point of view, almost all failures require only three or fewer nodes to reproduce, which is good news considering that these services typically run on a very large number of nodes. In addition, we found that a majority of the failures can simply be reproduced by unit tests even though conventional wisdom has it that failures that occur on a distributed system in production are extremely hard to reproduce offline. Nevertheless, we found the failure manifestations are generally complex, typically requiring multiple input events occurring in a specific order.

The 198 randomly sampled, real world, user-reported failures we studied are from the issue tracking databases of five popular distributed data-analytic and storage systems: Cassandra, HBase, HDFS, Hadoop MapReduce, and Redis. 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.

Software	**Language**	**Failures**		
		Total	Sampled	Catastrophic
Cassandra	Java	3,923	40	2
HBase	Java	5,804	41	21
HDFS	Java	2,828	41	9
MapReduce	Java	3,469	38	8
Redis	C	1,192	38	8
Total	–	17,216	198	48

Table 1: Number of reported and sampled failures for the systems we studied, and the catastrophic ones from the sample set



Yongle Zhang is a graduate student in computer engineering at the University of Toronto. His research interests are in operating systems and log analysis. He previously received an M.E. and B.E. in computer science from the Institute of Computing Technology of the Chinese Academy of Sciences and at Shandong University, respectively. yongle.zhang@mail.utoronto.ca

Pranay U. Jain is a final year undergraduate in computer engineering at the University of Toronto. Previously, he interned with the Amazon Web Services team. pranay.ujain@mail.utoronto.ca

Michael Stumm is a professor in the Electrical and Computer Engineering Department of the University of Toronto. He works in the general area of computer systems software with an emphasis on operating systems for distributed systems and multiprocessors. He co-founded two companies, SOMA Networks and OANDA, a currency trading company. stumm@ece.utoronto.ca

Table 1 shows the distribution of the failure sets. 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 further studied the characteristics of a specific subset of failures—the *catastrophic failures*, which we define as those 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 service providers, and they are not supposed to occur, considering these distributed systems are designed to withstand and automatically recover from component failures.

General Findings

What follows is a list of all of our general findings. Overall, our findings indicate that the failures are relatively complex, but they identify a number of opportunities for improved testing and diagnosis. Note that we only discuss the first five of the general findings in this article. Our OSDI paper [6] contains detailed discussions on the other general findings, and findings for catastrophic failures are discussed below (Findings 11-13).

1. A majority (77%) of the failures require more than one input event to manifest.
2. A significant number (38%) of failures require input events that typically occur only on long running systems.
3. The specific order of events is important in 88% of the failures that require multiple input events.
4. Twenty-six percent of the failures are non-deterministic—they are not guaranteed to manifest given the right input event sequences.
5. Almost all (98%) of the failures are guaranteed to manifest on no more than three nodes.
6. Among the non-deterministic failures, 53% have timing constraints only on the input events.
7. Seventy-six percent of the failures print explicit failure-related error messages.
8. For a majority (84%) of the failures, all of their triggering events are logged.
9. Logs are noisy: the median of the number of log messages printed by each failure is 824.
10. A majority (77%) of the production failures can simply be reproduced by a unit test.

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 three.*

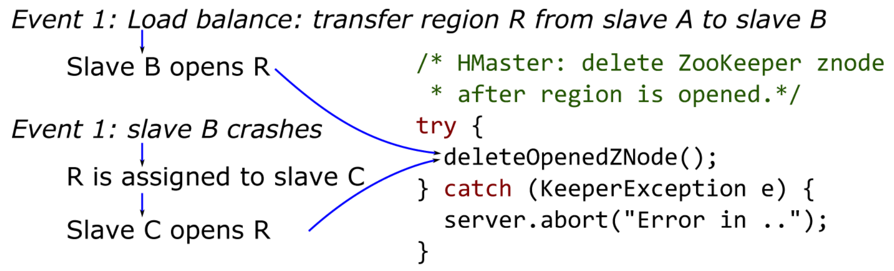
Figure 1 provides an example where two input events, a load balance event and a node crash, are required to take down the cluster. Note that we consider the events to be "input events" from a testing and diagnostic point of view—some of the events (e.g., "load balance" and "node crash") are not strictly user inputs but can easily be emulated in testing.

Finding 2: *A significant number (38%) of failures require input events that typically occur only on long running systems.*

The load balance event in Figure 1 is such an example. This finding suggests that many of these failures can be hard to expose during normal testing unless such events are intentionally exercised by testing tools.

Finding 3: *The specific order of events is important in 88% of the failures that require multiple input events.*

Consider again the example shown in Figure 1. The failure only manifests when the load balance event occurs before the crash of slave B. A different event order will not lead to failure.



```
/* HMaster: delete ZooKeeper znode
 * after region is opened.*/
try {
  deleteOpenedZNode();
} catch (KeeperException e) {
  server.abort("Error in ..");
}
```

Figure 1: A failure in HBase that requires two input events to trigger. A load balance event first causes a region R to be transferred from an overloaded slave A to a more idle slave B. After B opens R, HMaster deletes the ZooKeeper znode that is used to indicate R is being opened. If slave B crashes at this moment, another slave C is assigned to serve the region. After C opens R, HMaster tries to delete the same ZooKeeper znode again, but deleteOpenedZNode() throws an exception because the znode is already deleted. This exception takes down the entire cluster.

In many cases, even with the right combination and sequence of input events the failure is not guaranteed to manifest:

Finding 4: *Twenty-six percent of the failures are non-deterministic—they are not guaranteed to manifest given the right input event sequences.*

In these cases, additional timing relationships are required for the failures to manifest. For example, the failure in Figure 1 can only manifest when slave B crashes after the znode is deleted. If it crashes before the HMaster deletes the znode, the failure would not be triggered.

Findings 1–4 show the complexity of failures in large distributed systems. To expose the failures in testing, we need to not only explore the combination of multiple input events from an exceedingly large event space with many only occurring on long running systems, we also need to explore different permutations. Some further require additional timing relationships.

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 5: *Almost all (98%) of the failures are guaranteed to manifest on no more than three nodes.*

The number is similar for catastrophic failures, where 98% of them manifest on no more than three nodes. Finding 5 implies that it is not necessary to have a large cluster to test for and reproduce failures.

Note that Finding 5 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.

Catastrophic Failures

Table 1 shows that 48 failures in our 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 a cluster-wide outage, a hung cluster, or a loss to all or to a majority of the user data.

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 failure, automatically elect a new master node and fail over to it. Cassandra is a peer-to-peer system and thus by design avoids single points of failure. Then why do catastrophic failures still occur?

Finding 11: *Almost all catastrophic failures (92%) are the result of incorrect handling of non-fatal errors explicitly signaled in software* (see Figure 2).

These catastrophic failures are the result of more than one fault triggering, where the initial fault, whether due to hardware, misconfiguration, or 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.

Overall, we found that the developers are good at anticipating possible errors. In all but one case, the errors were properly checked for in the software. However, we found the developers were often negligent in handling these errors. This is further

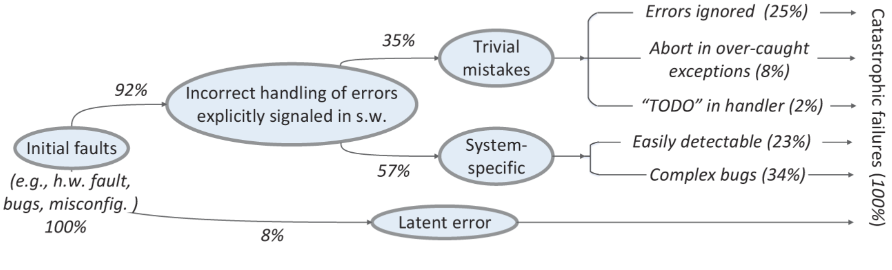

Initial faults
(e.g., h.w. fault, bugs, misconfig.)
100%
92%
Incorrect handling of errors explicitly signaled in s.w.
35%
Trivial mistakes
57%
System-specific
8%
Latent error
Errors ignored (25%)
Abort in over-caught exceptions (8%)
"TODO" in handler (2%)
Easily detectable (23%)
Complex bugs (34%)
Catastrophic failures (100%)

Figure 2: Breakdown of all catastrophic failures by their error handling

corroborated in Findings 12 and 13, 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 (as evidenced by the long uptime of these systems).

Nevertheless, the correctness of error handling code is particularly important given their impact. Previous studies [4, 5] 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 all of them [1]. It is therefore unavoidable that some of these faults will manifest themselves into errors, and error handling then becomes the last line of defense [3].

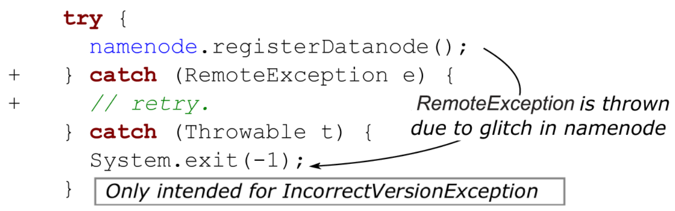
Trivial Mistakes in Error Handlers

Finding 12: *Thirty-five percent 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 2 breaks down the trivial mistakes into three categories: (1) the error handler ignores explicit errors; (2) the error handler over-catches an exception and aborts the system; and (3) the error handler contains “TODO” or “FIXME” comments.

Twenty-five percent of the catastrophic failures were caused by ignoring explicit errors. (An error handler that only logs the error is also considered to be ignoring the error.) For systems written in Java, the exceptions were all explicitly thrown, whereas in Redis they were system call error returns.

Another 8% of the catastrophic failures were caused by developers prematurely aborting the entire cluster on a non-fatal exception. While in principle one would need system-specific knowledge to determine when to bring down the entire cluster, the aborts we observed were all within *exception over-catches*, where a higher level exception is used to catch multiple different lower-level exceptions. Figure 3 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 brought down every datanode. The fix is to handle RemoteException explicitly.

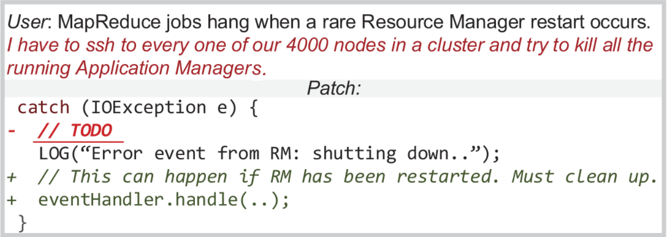


```
      try {
        namenode.registerDatanode();
+     } catch (RemoteException e) {
+       // retry.
      } catch (Throwable t) {
        System.exit(-1);
      }
```

RemoteException is thrown due to glitch in namenode

Only intended for IncorrectVersionException

Figure 3: An entire HDFS cluster brought down by an over-catch



User: MapReduce jobs hang when a rare Resource Manager restart occurs. *I have to ssh to every one of our 4000 nodes in a cluster and try to kill all the running Application Managers.*

Patch:

```
 catch (IOException e) {
-   // TODO
    LOG("Error event from RM: shutting down..");
+   // This can happen if RM has been restarted. Must clean up.
+   eventHandler.handle(..);
 }
```

Figure 4: A catastrophic failure in MapReduce where developers left a “TODO” in the error handler

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

System-Specific Bugs

Fifty-seven percent of catastrophic failures are caused by incorrect error handling where system-specific knowledge is required to detect the bugs (see Figure 2).

Finding 13: *In 23% of the catastrophic failures, the mistakes in error handling were system specific, but were 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 faulty and simply never tested. The failure shown in Figure 1 belongs to this category. Once a test case can deterministically trigger the catch block, the failure will manifest 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. 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. 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.

The remaining 34% of catastrophic failures involve complex bugs in the error handling logic. 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.

Aspirator: A Simple Checker

In the subsection “Trivial Mistakes in Error Handlers,” we observed that some of the most catastrophic failures are caused by trivial mistakes that fall into three simple categories: (1) error handlers that are empty or only contain log printing statements;

(2) error handlers that over-catch exceptions and abort; and (3) error handlers that contain phrases like "TODO" and "FIXME." We built a rule-based static checker, Aspirator, capable of locating these bug patterns. We provided two implementations of Aspirator: one as a stand-alone tool that analyzes Java bytecode, and another version that can be integrated with the Java build system to catch these bugs at compile-time. The implementation details of Aspirator can be found in our OSDI paper [6].

Checking Real-World Systems

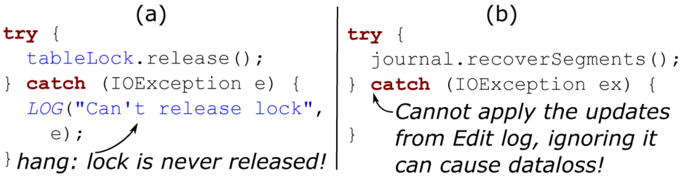
We first evaluated Aspirator on the set of catastrophic failures used in our study. If Aspirator had been used and the identified bugs fixed, 33% of the Cassandra, HBase, HDFS, and MapReduce's catastrophic failures we studied would have been prevented. We then used Aspirator to check the latest stable versions of these four systems plus five other systems: Cloudstack, Hive, Tomcat, Spark, and ZooKeeper.

We categorized each warning generated by Aspirator into one of three categories: bug, bad practice, and false positive. Bugs are the cases where the error handling logic will clearly lead to unexpected failures. False positives are those that clearly would not lead to a failure. Bad practices are cases that the error handling logic is suspicious of, but we could not definitively infer the consequences without domain knowledge. 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.

Overall, Aspirator detected 121 new bugs and 379 bad practices along with 115 false positives. Aspirator found new bugs in every system we checked.

Many bugs detected by Aspirator could indeed lead to catastrophic failures. For example, all four bugs caught by the abort-in-over-catch checker could bring down the cluster on an unexpected exception similar to the one in Figure 3. They have all been fixed by the developers of the respective systems.

Some bugs can also cause the cluster to hang. Aspirator detected five bugs in HBase and Hive that have a pattern similar to the one depicted in Figure 5(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.



(a)

```
try {
  tableLock.release();
} catch (IOException e) {
  LOG("Can't release lock",
    e);
}
```

hang: lock is never released!

(b)

```
try {
  journal.recoverSegments();
} catch (IOException ex) {
}
```

Cannot apply the updates from Edit log, ignoring it can cause dataloss!

Figure 5: Two new bugs found by Aspirator

Figure 5(b) shows a bug that could lead to data loss. An IOException could be thrown when HDFS is recovering the updates from the edit log. Ignoring this exception could cause a silent data loss.

Experience

Interaction with developers: We reported 171 bugs and bad practices to the developers of the respective systems: 143 have already been confirmed or fixed by the developers, 17 were rejected, and developers never responded to the other 11 reports.

We received mixed feedback from developers. On the one hand, positive comments include: *"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 why developers may be oblivious to the handling of errors. First, some errors are ignored because they are not regarded as critical at the time, and the importance of the error handling is realized only when the system suffers a serious failure. 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.

Second, developers may believe the errors would never (or only very rarely) occur. Consider the following code snippet detected by Aspirator from HBase:

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

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 containing similar comments in the systems we checked. We argue that errors that "can never happen" should be handled defensively to prevent them from propagating. This is because developers' judgment 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' judgment was indeed wrong. The constructor is implemented as follows:

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

where an IOException is thrown on an integer overflow; yet swallowing this exception could lead to a data loss. The developers later fixed this by handling the IOException properly.

Third, 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 poorly documented third-party components. 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:

```
} catch (NoTransitionException ne) {
  /  Why this can happen? Ask God not me.  /
}
```

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

Finally, feature development is often prioritized over exception handler coding when release deadlines loom. We embarrassingly experienced this ourselves when we ran Aspirator on Aspirator's code: We found five 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 nine 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 five systems we studied, Cassandra also has the lowest rate of catastrophic failures in its randomly sampled failure set (see Table 1).

Reactions from HBase developers: Our OSDI paper prompted HBase developers to start the initiative to fix all the existing bad practices. They intend to use Aspirator as their compile-time checker [2].

Conclusions

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

Existing testing techniques will find it difficult to successfully uncover many of these error-handling bugs. They all use a "top-down" approach: start the system using generic inputs and actively inject errors at different stages. However, the size of the input and state space makes the problem of exposing these bugs intractable.

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) purposefully construct test cases 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/.

Acknowledgments

We greatly appreciate the anonymous OSDI reviewers, Jason Flinn, Leonid Ryzhyk, Ashvin Goel, David Lie, and Rik Farrow for their insightful feedback. We thank Dongcai Shen for help with reproducing five bugs. This research is supported by an NSERC Discovery grant, NetApp Faculty Fellowship, and Connaught New Researcher Award.

References

[1] J. Dean, "Underneath the Covers at Google: Current Systems and Future Directions," in *Google I/O,* 2008.

[2] HBase-12187: review in source the paper "Simple Testing Can Prevent Most Critical Failures": https://issues.apache.org/jira/browse/HBASE-12187.

[3] P.D. Marinescu and G. Candea, "Efficient Testing of Recovery Code Using Fault Injection," *ACM Transaction on Computer Systems,* vol. 29, no. 4, Dec. 2011.

[4] D. Oppenheimer, A. Ganapathi, and D.A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems, USITS '03,* 2003, pp. 1–15.

[5] A. Rabkin and R. Katz, "How Hadoop Clusters Break," *Software, IEEE,* vol. 30, no. 4, 2013, pp. 88–94.

[6] D. Yuan, Y. Luo, X. Zhuang, G.R. Rodrigues, X. Zhao, Y. Zhang, P.U. Jain, and M. Stumm, "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI '14,* 2014, pp. 249–265.