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, FourSqure, 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[1] 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 misconfiguraitons [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 organization. But it does suggest that top-down testing, say

---

[1]Throughout 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.

Software	lang.	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.

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

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

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

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.

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.

Num. of events	%	
1	23%	}single event
2	50%	
3	17%	
4	5%	}multiple events: 77%
> 4	5%	

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

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

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%.

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 into 9 categories. We consider these events to be "input events" from a testing and diagnostic point of view — some of the events (e.g., "unreachable node", "data corruption") are not strictly user inputs but can easily be emulated by a tester or testing tools. Note that many of the events have specific requirements for a failure to manifest (e.g., a "file write" event needs to occur on a particular data block), making the input event space to explore for testing immensely large.

Of the 23% of failures that require only a single event to manifest, the event often involves rarely used or newly introduced features, or are caused by concurrency bugs.

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
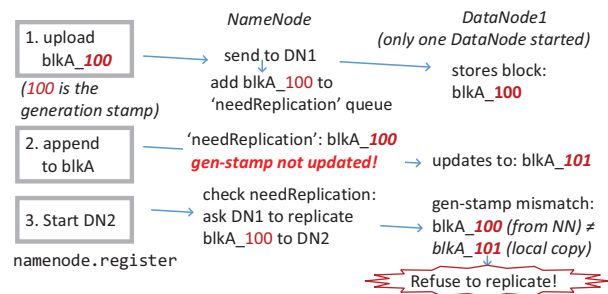

Figure 1: An HDFS failure where a data block remains under-replicated, potentially leading to a data loss. Three input events are needed (shown in boxes): (1) the user uploads a data block, causing HDFS to assign a generation stamp. NameNode (NN) asks DataNode1 (DN1) to store this block, and because this block is currently under-replicated, adds it to `needReplication` queue. (2) the user appends to this block, causing DN1 to increment the generation stamp from 100 to 101. However, the generation stamp in the `needReplication` queue is not updated – an error. (3) DN2 is started, so NN asks DN1 to replicate the block to DN2. But since the generation stamps from `needReplication` queue and DN1 do not match, DN1 keeps refusing to replicate.

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.

| Number | cumulative distribution function | |
of nodes	all failures	catastrophic
1	37%	43%
2	84%	86%
3	98%	98%
> 3	100%	100%

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

Software	num. of deterministic failures
Cassandra	76% (31/41)
HBase	71% (29/41)
HDFS	76% (31/41)
MapReduce	63% (24/38)
Redis	79% (30/38)
Total	74% (147/198)

Table 6: Number of failures that are deterministic.

Source of non-determinism	number
Timing btw. input event & internal exe. event	27 (53%)
Multi-thread atomicity violation	13 (25%)
Multi-thread deadlock	3 (6%)
Multi-thread lock contention (performance)	4 (8%)
Other	4 (8%)
Total	51 (100%)

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

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 inputing 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 configurations 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 simul-

taneous 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

HMaster   RegionServer (RS)

Cannot connct to RS,   ┌─────────────────────────┐
thinks RS is dead   ◁╌╌╌╌╌┤ 1. RS network error   │
       ╎   └───────────┬─────────────┘
       ▽   ▽
Recovers RS' region data   ┌─────────────────────────┐
from its HLog   │ 2. RS is back online   │
       ╎   └───────────┬─────────────┘
       ╎   ▽
       ╎   ┌─────────────────────────┐
Recovery finishes   │ 3. rolls its HLog (HBase's│
       ╎   │   write-ahead-log)   │
       ╎   └───────────┬─────────────┘
       ▽   ▽
A new RS is assigned to ◁──┤ 4. Receive client writes requests,│
serve the recovered region │   log into the new HLog │
                           └─────────────────────────┘

        Order dependency btw. input   Data writes
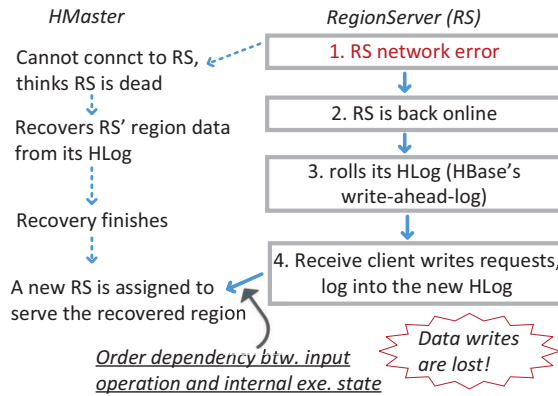   operation and internal exe. state   are lost!

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.

```
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 stem-
ming 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 de-
terministically reproduce the failure.

The majority of the remaining 24 non-deterministic
failures stem from shared-memory multi-threaded inter-
leavings. We observed three categories of concurrency
bugs in our dataset: atomicity violation [42], deadlock,
and lock contention that results in performance degra-
dation. It is much harder to expose and reproduce such
failures because it is hard for users or tools to control tim-
ing, 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
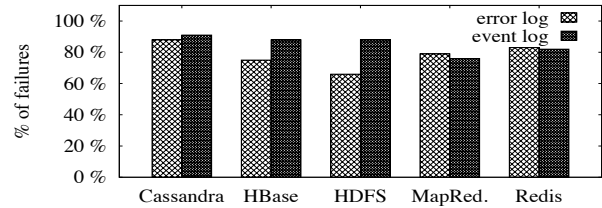(we had to manually introduce additional timing delays,

Figure 4: The logging comprehensiveness of the studied fail-
ures. Logging of both the input events and errors are consid-
ered. For failures requiring multiple events to trigger, we count
it as "logged" only when *all* the required events are logged.

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 sys-
tems, including Apache httpd, PostgreSQL, SVN, squid,
and GNU Coreutils, where only 43% of failures had ex-
plicit failure-related error messages logged. We surmise
there are three possible reasons why developers output
log messages more extensively for the distributed sys-
tems we studied. First, since distributed systems are
more complex, and harder to debug, developers likely
pay more attention to logging. Second, the horizon-
tal scalability of these systems makes the performance
overhead of outputting log message less critical. Third,
communicating through message-passing provides natu-
ral points to log messages; for example, if two nodes can-
not 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 re-
play 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 intru-
sive 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.

Software	% of failures reproducible by unit test
Cassandra	73% (29/40)
HBase	85% (35/41)
HDFS	82% (34/41)
MapReduce	87% (33/38)
Redis	58% (22/38)
Total	77% (153/198)

Table 8: Percentage of failures that can be reproduced by a unit test. The reason that only a relatively small number of Redis failures can be reproduced by unit tests is that its unit-test framework is not as powerful, being limited to command-line commands. Consequently, it cannot simulate many errors such as node failure, nor can it call some internal functions directly.

This number was obtained when reproducing 73 of the 198 failures with a *minimal* configuration and using a minimal workload that is just sufficient to reproduce the failure. Moreover, we did not count the messages printed during the start-up and shut-down phases.

This suggests that manual examination of the log files could be tedious. If a user only cares about the error symptoms, a selective `grep` on the error verbosity levels will reduce noise since a vast majority of the printed log messages are at INFO level. However, the input events that triggered the failure are often logged 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

```
public void testLogRollAfterSplitStart {
  startMiniCluster(3);
  // create an HBase cluster with 1 master and 2 RS

  HMaster.splitHLog();
  // simulate a hlog splitting (HMaster's recovery
  // of RS' region data) when RS cannot be reached

  RS.rollHLog();
  // simulate the region server's log rolling event

  for (i = 0; i < NUM_WRITES; i++)
    writer.append(..); // write to RS' region

  HMaster.assignNewRS();
  // HMaster assigns the region to a new RS

  assertEquals (NUM_WRITES, countWritesHLog());
  // Check if any writes are lost
}
```

Figure 6: Unit test for the failure shown in Figure 2.

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 failure, automatically elect a new master node and fail
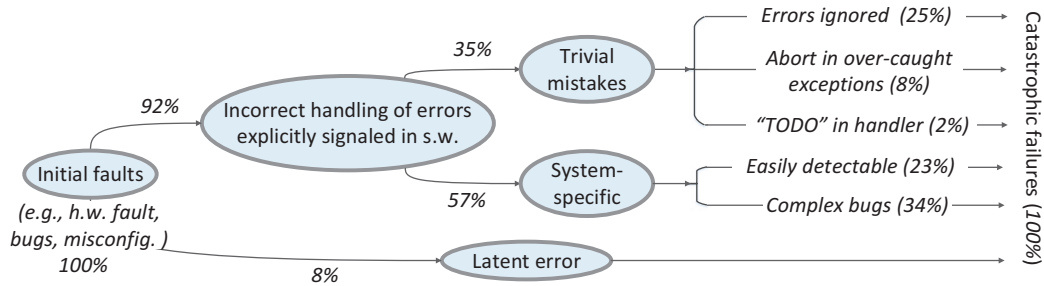
Figure 5: Break-down of all catastrophic failures by their error handling.

over to it.[2] 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

---
[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.

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.
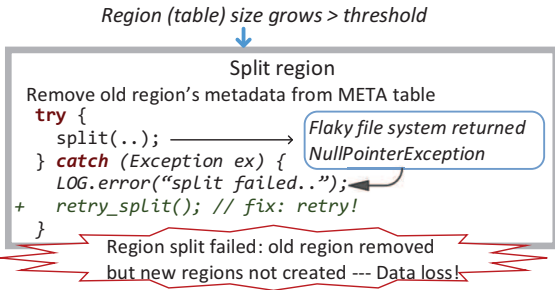
Region (table) size grows > threshold

```
                    Split region
  Remove old region's metadata from META table
  try {
      split(..);            →  Flaky file system returned
  } catch (Exception ex) {     NullPointerException
      LOG.error("split failed..");  ←
+     retry_split(); // fix: retry!
  }
      Region split failed: old region removed
      but new regions not created --- Data loss!
```

Figure 7: A data loss in HBase where the error handling was simply empty except for a logging statement. The fix was to retry in the exception handler.

```
      try {
          namenode.registerDatanode();
+     } catch (RemoteException e) {
+         // retry.                    RemoteExcepion is thrown
      } catch (Throwable t) {          due to glitch in namenode
          System.exit(-1);  ←
      }
      Only intended for IncorrectVersionException
```

Figure 8: 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 Manager.
                            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 9: A catastrophic failure in MapReduce where developers left a "TODO" in the error handler.

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-catch*, where a higher level exception is used to catch multiple different lower-level exceptions. 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.

```
1: HMaster assigns region R from RS1 to RS2
      ↓
2: RS2 opens R    HMaster.OpenedRegionHandler()
      ↓            try {
4: RS2 dies   ←   3: newlyOpened(getZookeeper(), R);
                    // inform zookeeper
5: HMaster          // that R is OPENED
assigns R to RS3  } catch (KeeperException e) {
      ↓             // R is already OPENED...
6: RS3 opens R →  7:abort("Error in .. ..");
                  }
```
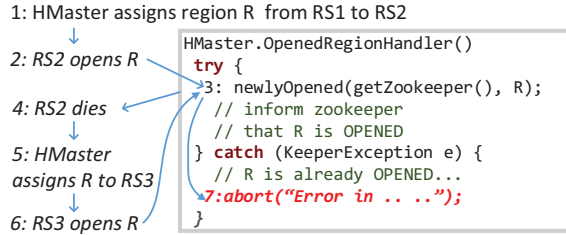
Figure 10: A catastrophic failure where the error handling code was wrong and simply not tested at all. A rare sequence of events caused `newlyOpened()` to throw a rare KeeperException, which simply took down the entire HBase cluster.

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 Fate&Destini [23] are intelligent to inject errors only at appropriate points and avoid duplicated injections. Such techniques inevitably have greatly
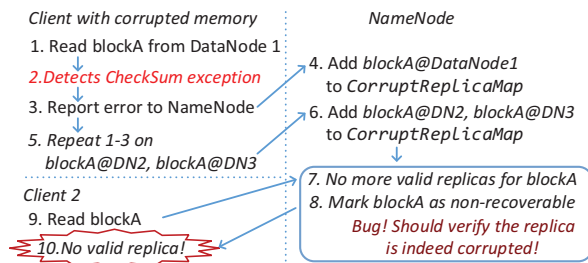
Client with corrupted memory

1. Read blockA from DataNode 1

2. Detects CheckSum exception

3. Report error to NameNode

5. Repeat 1-3 on
*   blockA@DN2, blockA@DN3*

Client 2

9. Read blockA

10. No valid replica!

NameNode

4. Add *blockA@DataNode1*
   to *CorruptReplicaMap*

6. Add *blockA@DN2, blockA@DN3*
   to *CorruptReplicaMap*

7. No more valid replicas for blockA

8. Mark blockA as non-recoverable

Bug! Should verify the replica
is indeed corrupted!

Figure 11: A massive data loss for all clients in HDFS. A client with corrupted RAM reported data corruption on almost every block it reads to the namenode. Instead of verifying the checksum on datanodes, namenode blindly trusts the faulty client and marks the blocks as permanently corrupted, causing a massive data loss to all clients.

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. Therfore, 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 system, 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 */ }
if (uri == null) { // handle it here!
```

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 instruction 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 programmers to specify exceptions that should not result in a warning. In our testing, we ignored all instances of the FileNotFound exception, because we found the vast majority 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 significantly reduce the number of false positives.

Limitations: As a proof-of-concept, Aspirator currently only works on Java and other languages that are compatible with Java bytecode (e.g., Scala), where exceptions are supported by the language and are required to be explicitly 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 easily 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,


```
           (a)                        (b)
try {                      try {
   tableLock.release();       journal.recoverSegments();
} catch (IOException e) {   } catch (IOException ex) {
   LOG("Can't release lock",    Cannot apply the updates
      e);                       from Edit log, ignoring it
} hang: lock is never released!    can cause dataloss!
                           }
```

Figure 12: Two new bugs found by Aspirator.

some false positives are emitted. However, because Aspirator provides, with each warning, a list of caught exceptions 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 probably 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 studied could have been prevented.

We then used Aspirator to check the latest stable versions of 9 distributed systems or components used to build distributed systems (e.g., Tomcat web-server). Aspirator's analysis finishes within 15 seconds for each system 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 categorized 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 knowledge. 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

System	Handler blocks	Bug				Bad practice				False pos.
		total / confirmed		ignore / abort / todo		total / confirmed		ignore / abort / todo		
Cassandra	4,365	2	2	2	-	-	2	2	2	-	-	9
Cloudstack	6,786	27	24	25	-	-	185	21	182	1	2	20
HDFS	2,652	24	9	23	-	1	32	5	32	-	-	16
HBase	4,995	16	16	11	3	2	43	6	35	5	3	20
Hive	9,948	25	15	23	-	2	54	14	52	-	2	8
Tomcat	5,257	7	4	6	1	-	23	3	17	4	2	30
Spark	396	2	2	-	-	2	1	1	1	-	-	2
YARN/MR2	1,069	13	8	6	-	7	15	3	10	4	1	1
Zookeeper	1,277	5	5	5	-	-	24	3	23	-	1	9
Total	36,745	**121**	**85**	101	4	16	**379**	**58**	354	14	11	115

Table 9: Results of applying Aspirator to 9 distributed systems. If a case belongs to multiple categories (e.g., an empty handler may also contain a "TODO" comment), we count it only once as an ignored exception. The "Handler blocks" column shows the number of exception handling blocks that Aspirator discovered and analyzed. "-" indicates Aspirator reported 0 warning.

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:

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

In addition to FileNotFound 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:

```
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 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:

```
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:

```
} 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, 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 duplicated 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 suggests 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/`.

Acknowledgements

References

[1] Why Amazon's cloud titanic went down. `http://money.cnn.com/2011/04/22/technology/amazon_ec2_cloud_outage/index.htm`.

[2] Apache Cassandra. `http://cassandra.apache.org`.

[3] Apache HBase. `http://hbase.apache.org`.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[5] I. Beschastnikh, Y. Brun, , J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *Proceedings*

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

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. Mining temporal invariants from partially ordered logs. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML'11, pages 3:1–3:10, 2011.

[7] J. Bloch. *Effective Java (2nd Edition)*. Prentice Hall, 2008.

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[9] Chaos monkey. `https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey`.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, Feb. 2012.

[11] Chord: A program analysis platform for Java. `http://pag.gatech.edu/chord`.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, 2001.

[13] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405, 2013.

[14] J. Dean. Underneath the covers at Google: current systems and future directions. In *Google I/O*, 2008.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, OSDI'02, 2002.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation*, OSDI'00, pages 1–16, 2000.

[17] Facebook: More details on today's outage. `https://www.facebook.com/note.php?note_id=431441338919&id=9445547199&ref=mf`.

[18] Hadoop team. Fault injection framework: How to use it, test using artificial faults, and develop new faults. `http://wiki.apache.org/hadoop/HowToUseInjectionFramework`.

[19] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd USENIX Symposium on Networked System Design and Implementation*, NSDI'05, 2005.

[20] C. Fu and G. B. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *29th International Conference on Software Engineering*, ICSE'07, pages 230–239, 2007.

[21] Google outage reportedly caused big drop in global traffic. `http://www.cnet.com/news/google-outage-reportedly-caused-big-drop-in-global-traffic/`.

[22] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[23] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: a framework for cloud recovery testing. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[24] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, 2008.

[25] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 265–278, October 2011.

[26] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[27] Hadoop Distributed File System (HDFS) architecture guide. `http://hadoop.apache.org/docs/stable/hdfs_design.html`.

[28] Hadoop MapReduce. `http://hadoop.apache.org/docs/stable/mapred_tutorial.html`.

[29] Hadoop market is expected to reach usd 20.9 billion globally in 2018. `http://www.prnewswire.com/news-releases/hadoop-market-is-expected-to-reach-usd-209-billion-globally-in-2018-transparency-market-research-217735621.html`.

[30] DB-Engines ranking of wide column stores. `http://db-engines.com/en/ranking/wide+column+store`.

[31] HBase bug report 10452 – Fix bugs in exception handler. `https://issues.apache.org/jira/browse/HBASE-10452`.

[32] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notice*, 39(12):92–106, Dec. 2004.

[33] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 conference*, SIGCOMM '09, pages 243–254, 2009.

[34] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: finding liveness bugs in systems code. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation*, pages 243–256, April 2007.

[35] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity

multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 155–166, 2010.

[36] J.-C. Laprie. Dependable computing: concepts, limits, challenges. In *Proceedings of the 25th International Conference on Fault-tolerant Computing*, FTCS'95, pages 42–54, 1995.

[37] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation*, OSDI'14, 2014.

[38] S. Li, T. Xiao, H. Zhou, H. Lin, H. Lin, W. Lin, and T. Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proc. International Conference on Software Engineering (ICSE 2013), Software Engineering in Practice (SEIP) track*, May 2013.

[39] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation*, OSDI'04, 2004.

[40] G. C. Lorenzo Keller, Prasang Upadhyaya. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings International Conference on Dependable Systems and Networks*, DSN'08, 2008.

[41] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 31–44, 2013.

[42] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS'08, pages 329–339, 2008.

[43] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 conference*, SIGCOMM '02, pages 3–16, 2002.

[44] P. D. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *Proceedings of the 2010 USENIX annual technical conference*, USENIX ATC'10, 2010.

[45] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4):11:1–11:38, Dec. 2011.

[46] Missing dot drops Sweden off the internet. `http://www.networkworld.com/community/node/46115`.

[47] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution effiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[48] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[49] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, OSDI'04, 2004.

[50] 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 conference on USENIX Symposium on Internet Technologies and Systems*, USITS'03, pages 1–15, 2003.

[51] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 305–318, 2011.

[52] A. Rabkin and R. Katz. How Hadoop clusters break. *Software, IEEE*, 30(4):88–94, 2013.

[53] DB-Engines ranking of key-value stores. `http://db-engines.com/en/ranking/key-value+store`.

[54] Redis: an open source, advanced key-value store. `http://redis.io/`.

[55] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 270–280, 2009.

[56] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[57] C. Spatz. Basic statistics, 1981.

[58] M. Sullivan and R. Chillarege. Software defects and their impact on system availability — A study of field failures in operating systems. In *Twenty-First International Symposium on Fault-Tolerant Computing*, FTCS'91, pages 2–9, 1991.

[59] Summary of the Amazon EC2 and RDS service disruption. `http://aws.amazon.com/message/65648/`.

[60] The curse of the swallowed exception. `http://michaelscharf.blogspot.ca/2006/09/dont-swallow-interruptedexception-call.html`.

[61] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, 2011.

[62] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 193–204, New York, NY, USA, 2010. ACM.

[63] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the 24th ACM Sympo-*

sium on Operating Systems Principles, SOSP '13, 2013.

[64] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, 2009.

[65] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, April 2009.

[66] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, 2011.

[67] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation*, OSDI'12, pages 293–306, 2012.

Simple testing can prevent most critical failures
-- An analysis of production failures in distributed data-intensive systems

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

University of Toronto

Code and dataset:
http://www.eecg.toronto.edu/failureAnalysis/

UNIVERSITY OF
TORONTO

Key findings

- Failures are the results of complex sequence of events

- Catastrophic failures are caused by incorrect error handling
    - Many are caused by a small set of trivial bug patterns

- *Aspirator*: a simple rule-based static checker
    - Found 143 confirmed new bugs and bad practices

Distributed system failures can be deadly

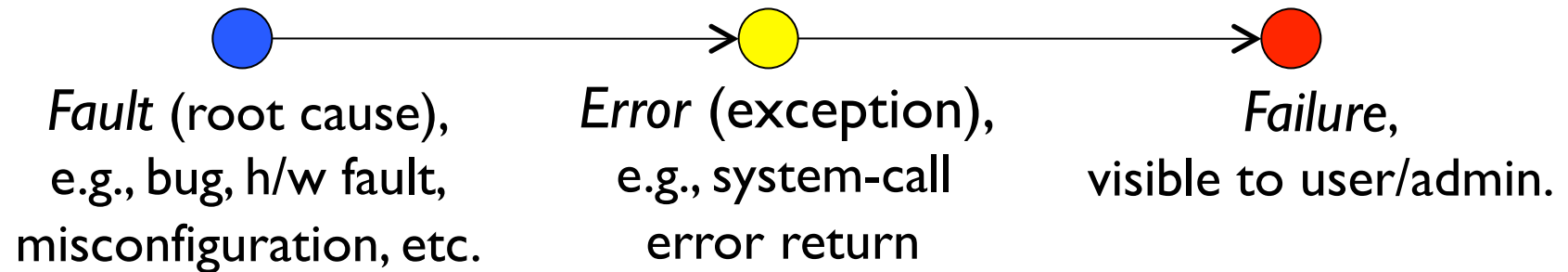**Amazon AWS outage downs Reddit, Quora, Foursquare, Instagram, NetFlix, and about 70 other sites.**

Google outage: Internet traffic plunges 40%.

Facebook goes down; users called 911.

A thorough analysis of real-world failures

▸ Study end-to-end failure propagation sequence

Fault (root cause),
e.g., bug, h/w fault,
misconfiguration, etc.

Error (exception),
e.g., system-call
error return

Failure,
visible to user/admin.

▸ Reveal the minimum conditions to expose failures
▸ Reveal the *weakest link*

▸ Previous works only studied elements in isolation

Study methodology

- Randomly sampled 198 user-reported failures*
  - Carefully studied the discussion and related code/patch
  - Reproduced 73 to understand them
- 48 are catastrophic --- they affect all or a majority of users

Software	Program language	Sampled failures	
		Total	Catastrophic
Cassandra	Java	40	2
HBase	Java	41	21
HDFS	Java	41	9
Hadoop MapReduce	Java	38	8
Redis	C	38	8
Total	-	198	48

* Analysis of each failure can be found at: http://www.eecg.toronto.edu/failureAnalysis/

Outline

➡ Failures are the results of complex sequence of events

▸ Catastrophic failures are caused by incorrect error handling

  ▸ Many are caused by *trivial* bugs

▸ *Aspirator*: a simple rule-based static checker

An example

User: "Sudden outage on the entire HBase cluster."

Event 1: Load balance: transfer Region R from slave A to B

Slave B opens R

Event 2: Slave B dies

R is assigned to slave C

Slave C opens R

```
/* Master: delete the
 * ZooKeeper znode after
 * the region is opened */
try {
  deleteZNode();
} catch (KeeperException e) {
  cluster.abort("…");
}
```

Not handled properly

Finding I: *multiple* events are required

77% of the failures require more than one input events

Only occur on long-running system (38%)

Event 1: *Load balance: transfer Region R from slave A to B*

↓

Slave B opens R

Event 2: *Slave B dies*

↓

R is assigned to slave C

↓

Slave C opens R

```
/* Master: delete the
 * ZooKeeper node after
 * the region is opened */
try {
  deleteZNode();
} catch (KeeperException e) {
  cluster.abort("…");
}
```

Finding II: event order matters

Order of events is important in 88% of the multi-events failures

Event 1: *Load balance: transfer Region R from slave A to B*

Slave B opens R

Event 2: *Slave B dies*

R is assigned to slave C

Slave C opens R

```
/* Master: delete the
 * ZooKeeper node after
 * the region is opened */
try {
deleteZNode();
} catch (KeeperException e) {
    cluster.abort("…");
}
```

9

Finding III: timing matters

26% of the failures are non-deterministic

Event 1: Load balance: transfer Region R from slave A to B

↓

Slave B opens R

Event 2: Slave B dies

↓

R is assigned to slave C

↓

Slave C opens R

```
/* Master: delete the
 * ZooKeeper node after
 * the region is opened */
try {
    deleteZNode();
} catch (KeeperException e) {
    cluster.abort("…");
}
```

Complexity is not surprising

▶ **These systems undergo thorough testing**

  ▶ Must provide unit test for every patch

  ▶ Use static checker on every check-in

  ▶ Use fault injection testing [HadoopFaultInjection]

▶ **Designed to provide high availability**

  ▶ E.g., automatic failover on master failures

Outline

▸ Failures are the results of complex sequence of events

⮕ **Catastrophic failures are caused by incorrect error handling**

   ▸ Catastrophic failures: those affect all or a majority of the users

▸ *Aspirator*: a rule-based static checker

Breakdown of catastrophic failures

92% of catastrophic failures are the result of incorrect error handling

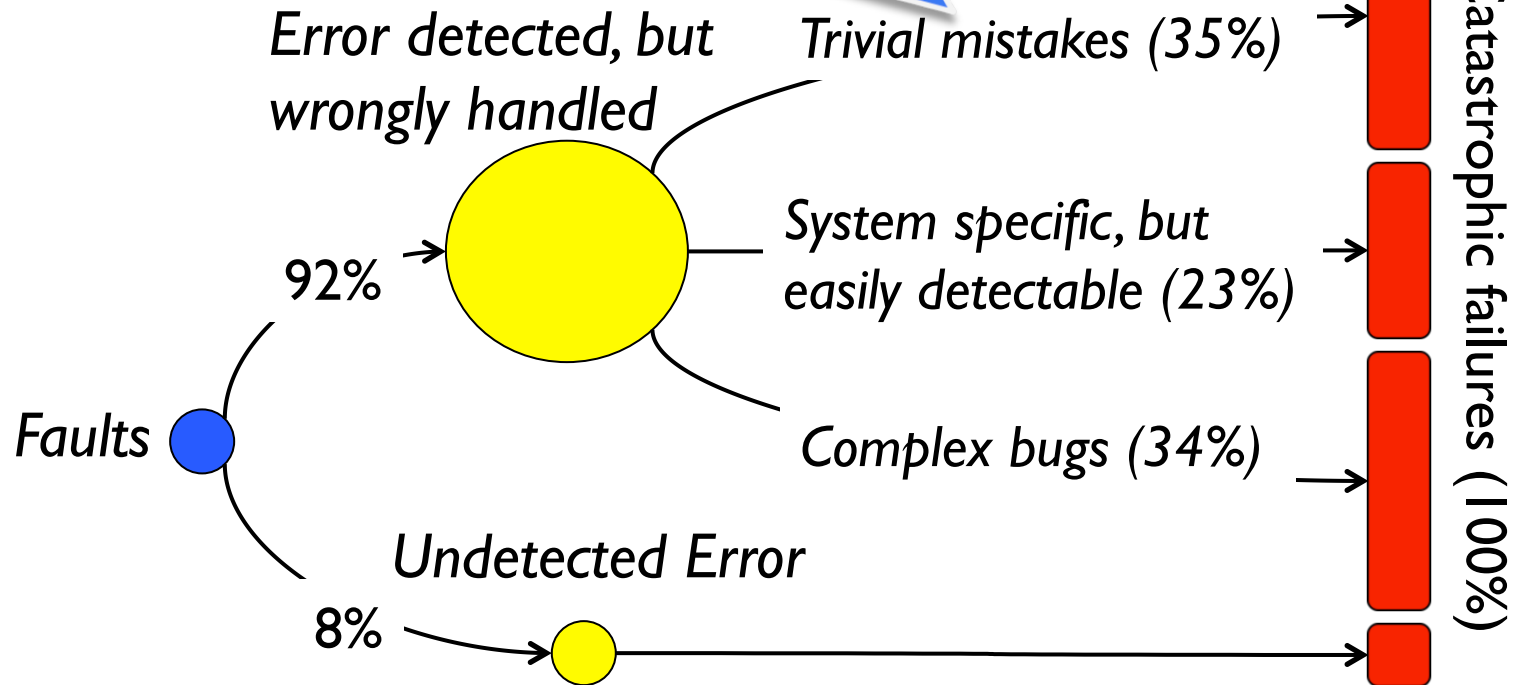▶ Error handling code is the *last line of defense* [Marinescu&Candea'11]

Error detected, but wrongly handled

92%

Faults

Undetected Error

8%

Catastrophic failures (100%)
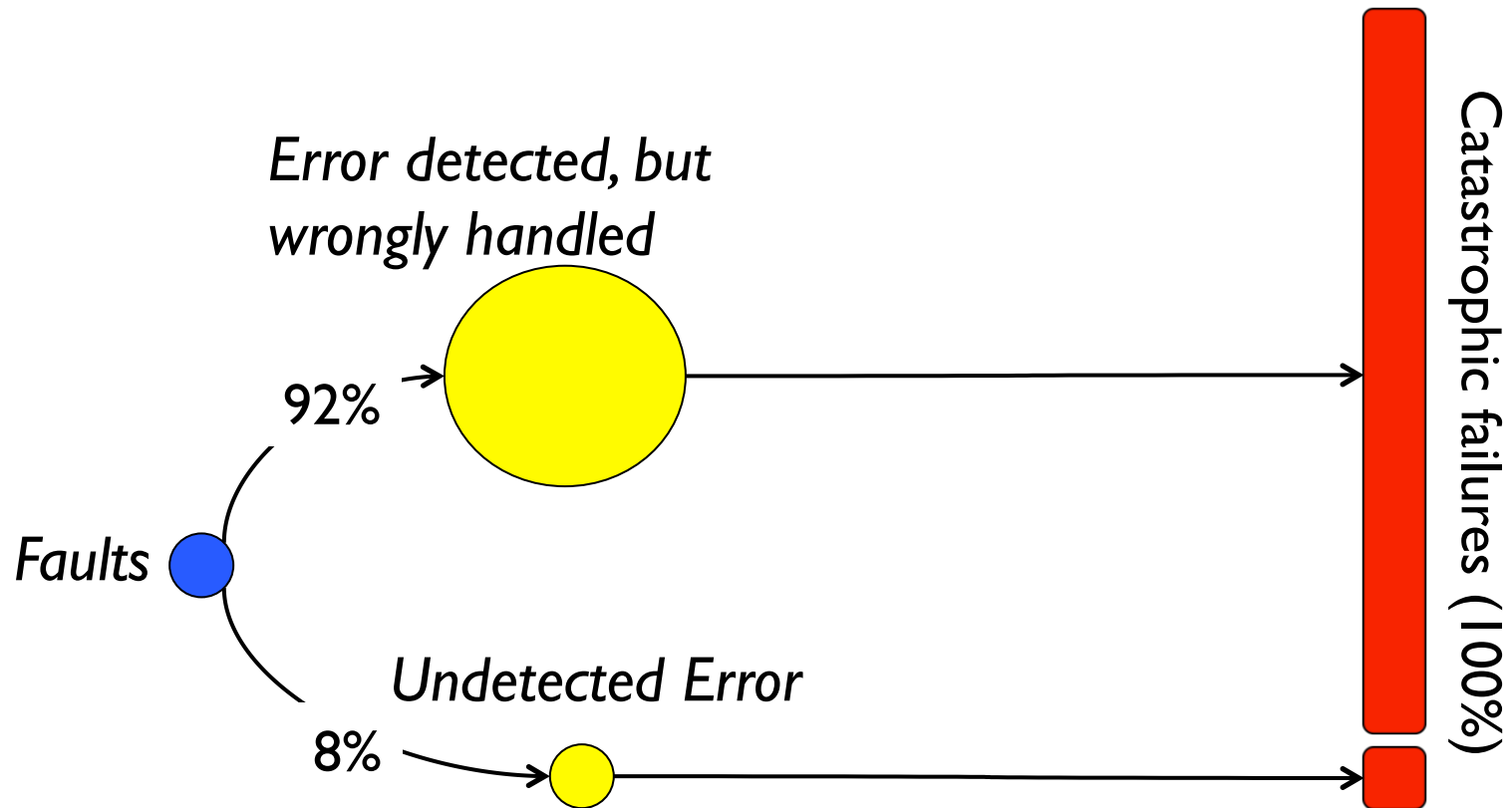
Trivial mistakes in error handling code

Example of abort in over-catch

```
      NonFatalException
      FatalException
} catch (Throwable t) {
  abort ("…");
}
```

Errors ignored (25%)
Abort in over-catch (8%)
"TODO" in handler (2%)

Error detected, but wrongly handled

Trivial mistakes (35%)

System specific, but easily detectable (23%)

92%

Complex bugs (34%)

Faults

Undetected Error

8%

Catastrophic failures (100%)

A failure caused by trivial mistake

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 kill all jobs.*"

```
catch (RebootException) {
  // TODO
  LOG("Error event from RM: shutting down...");
+ eventHandler.handle(exception_response);
}
```

Easily detectable bugs
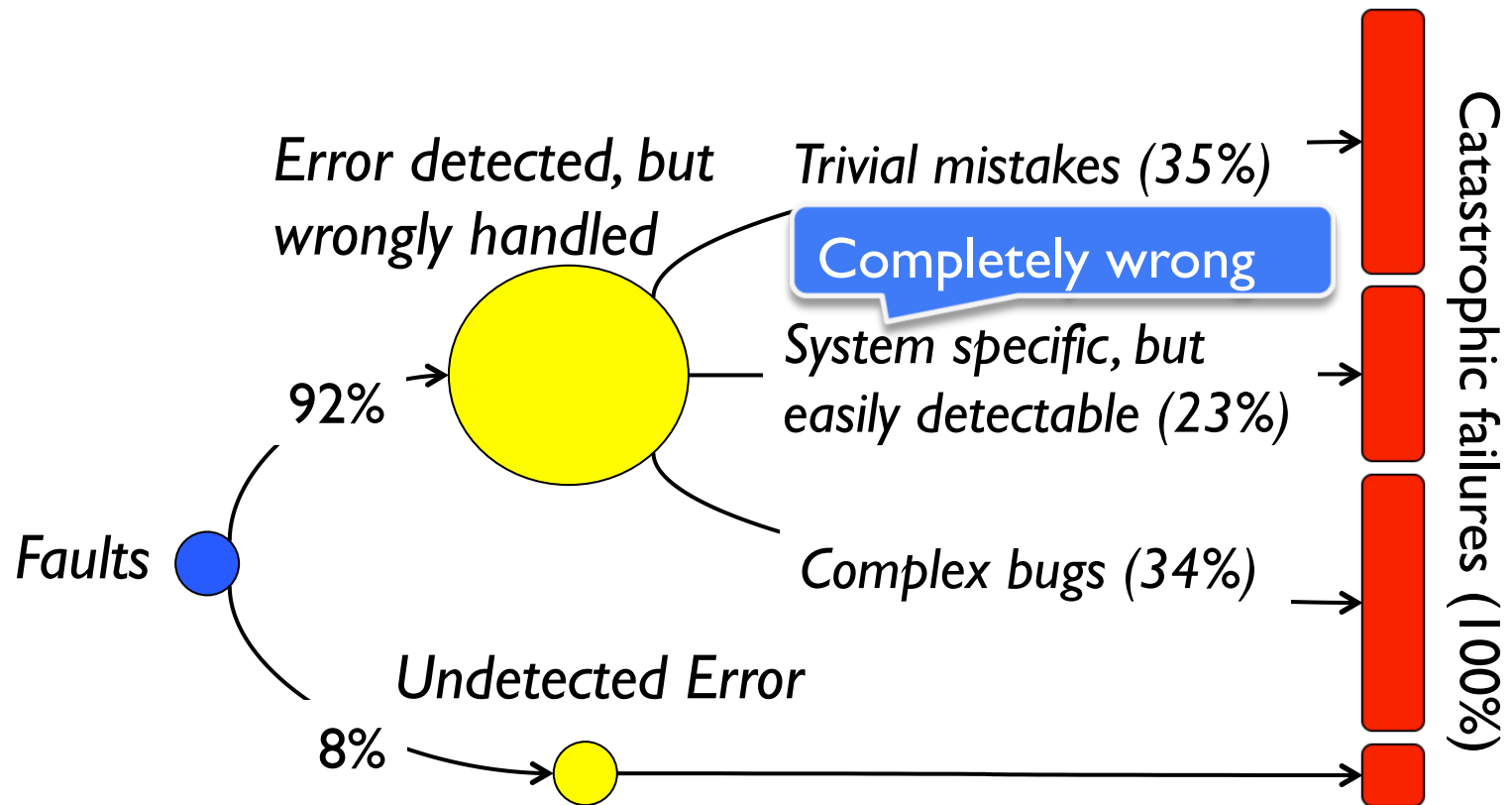
The HBase example: an easily detectable bug

▸ **Difficult to be triggered; easily detectable by code review**

Event 1: Load balance: transfer Region R from slave A to B

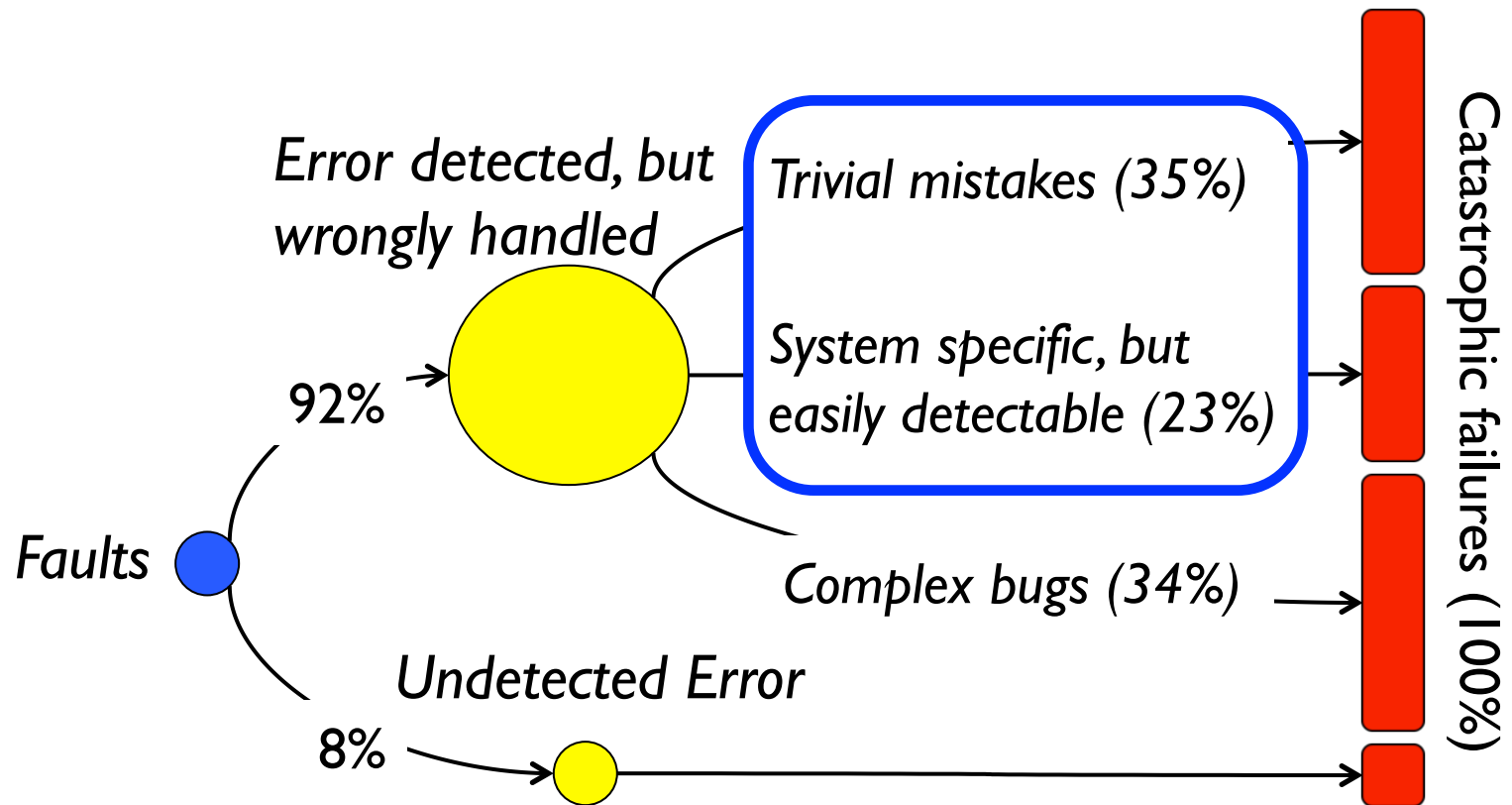Slave B opens R

Event 2: Slave B dies

R is assigned to slave C

Slave C opens R

```
/* Master: delete the
 * ZooKeeper znode after
 * the region is opened */
try {
  deleteZNode();
} catch (KeeperException e) {
  cluster.abort("…");
}
```

Completely wrong

Over half are trivial or easily detectable bugs

Outline

▸ Failures are the results of complex sequence of events

▸ Catastrophic failures are caused by incorrect error handling

➡ *Aspirator*: a simple rule-based static checker

Aspirator: a static checker for Java programs

- Three rules on exception handling
  - Not empty
  - Not abort on exception over-catch
  - No "TODO" or "FIXME" comment
- False positive suppression techniques (details in paper)

- Over 1/3 of catastrophic failures could have been prevented
  - If aspirator has been used and identified bugs fixed

All source code of aspirator is at: http://www.eecg.toronto.edu/failureAnalysis/

Checking real-world systems

new bugs in every system

	System	Bugs	Bad practice	False positive
	Cassandra	2	2	9
Training set	HBase	16	43	20
	HDFS	24	32	16
	Hadoop MapRed.2	13	15	1
	Cloudstack	27	185	20
	Hive	25	54	8
Testing set	Tomcat	7	23	30
	Spark	2	1	2
	Zookeeper	5	24	9
	Total	**121**	379	115

New bugs can lead to catastrophic failures

▸ Hang system

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

▸ Data loss

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

}
```

> Cannot recover updates from journal

▸ Cluster crash

▸ E.g., bugs found by "abort in over-catch" check

Mixed feedbacks from developers

▸ Reported 171 new bugs/bad practices

  ▸ 143 confirmed/fixed; 17 rejected; no response for the rest

"No one would have looked at this hidden feature; ignoring exceptions is bad precisely for this reason"

"I really want to fix issues in this line, because I really want us to use exceptions properly and never ignore them"

"I fail to see the reason to handle every exception."

Why do developers ignore error handling?

- **Developers think the errors *will never happen***
  - Code evolution may enable the errors
  - The judgment can be wrong

    ```java
    } catch (IOException e) {
      // will never happen
    }
    ```

- **Error handling is difficult**
  - Errors can be returned by 3rd party libraries

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

- **Feature development is prioritized**

Other findings in the paper

▸ Failures require no more than **3** nodes to manifest

▸ Failures can be reproduced offline by unit tests

   ▸ The triggering events are recorded in system log

▸ Non-deterministic failures can still be *deterministically reproduced*

Related work

- Error handling code is often buggy [Gunawi'08, Marinescu'10, Rubio-González'09, Sullivan'91, etc.]

- Studies on distributed system failures [Gray'85, Oppenheimer'03, Rabkin'13, etc.]

- Distributed system testing [ChaosMonkey, Gunawi'11, Guo'11, HadoopFaultInjection, Killian'07, Leesatapornwongsa'14, Yang'09, etc.]

Conclusions

▸ **Failures are the results of complex sequence of events**

▸ **Catastrophic failures are caused by incorrect error handling**

   ▸ Many are caused by a small set of trivial bug patterns

▸ *Aspirator*: a simple rule-based static checker

   ▸ Found 143 confirmed *new* bugs and bad practices

Unexpected fun: comments in error handlers

```
/* If this happens, hell will unleash on earth. */


/* FIXME: this is a buggy logic, check with alex. */


/* TODO: this whole thing is extremely brittle. */


/* TODO: are we sure this is OK? */


/* I really thing we should do a better handling of these
 * exceptions. I really do. */


/* I hate there was no piece of comment for code
 * handling race condition.
 * God knew what race condition the code dealt with! */
```

Source code and dataset:

http://www.eecg.toronto.edu/failureAnalysis/

Thanks!

UNIVERSITY OF
TORONTO