Simple Testing Can Prevent Most Critical Failures

An Analysis of Production Failures in Distributed Data-intensive Systems

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

By John Amadeo, Jessica Ambrosio, Ken Cheng, Daniel Ernst, Roland Huang, Eugene Szeto, and Angel Zhang
Roadmap

- Introduction
- Complexity of Failures And Opportunity for Improved Testing
- Role of Timing And Use of Logs in Reproducing Failures
- Catastrophic Failures
- Aspirator, A Simple Checker
- Comparisons: Aspirator and Other Test Suits
- Conclusion
Importance of Failures in Distributed Systems

An outage to AWS in 2011 brought down Reddit, Quora, FourSquare, and about 70 other sites.
Importance of Failures in Distributed Systems

An outage to AWS in 2011 brought down Reddit, Quora, FourSquare, and about 70 other sites.

Outage of Google in 2013 brought down internet traffic by 40%.
Importance of Failures in Distributed Systems

An outage to AWS in 2011 brought down Reddit, Quora, FourSquare, and about 70 other sites.

Outage of Google in 2013 brought down internet traffic by 40%.

Systems were designed to be highly available and were intensely tested, but still prone to failure.
Importance of Failures in Distributed Systems

An outage to AWS in 2011 brought down Reddit, Quora, FourSquare, and about 70 other sites.

Outage of Google in 2013 brought down internet traffic by 40%.

Systems were designed to be highly available and were intensely tested, but still prone to failure.

Why do they still fail?
198 randomly sampled real-world failures

- HDFS, Hadoop MapReduce, HBase, Cassandra, Redis
  - Very popular systems
  - Representativeness
    - Only distributed, data-intensive software systems
    - Diverse types of programs: Master-slave and P2P, Java and C
Methodology

198 randomly sampled real-world failures

- HDFS, Hadoop MapReduce, HBase, Cassandra, Redis
  - Very popular systems
  - Representativeness
    - Only distributed, data-intensive software systems
    - Diverse types of programs: Master-slave and P2P, Java and C

- Extracted from issue tracking databases for the systems
  - More rigorously documented, less likely to be result of trivial mistake than user discussion forums
Methodology

198 randomly sampled real-world failures

- HDFS, Hadoop MapReduce, HBase, Cassandra, Redis
  - Very popular systems
  - Representativeness
    - Only distributed, data-intensive software systems
    - Diverse types of programs: Master-slave and P2P, Java and C

- Extracted from issue tracking databases for the systems
  - More rigorously documented, less likely to be result of trivial mistake than user discussion forums

- Only selected tickets for severe failures after 2010, where reporter and assignee are not the same person
Roadmap

- Introduction
- **Complexity of Failures And Opportunity for Improved Testing**
- Role of Timing And Use of Logs in Reproducing Failures
- Catastrophic Failures
- Aspirator, A Simple Checker
- Comparisons: Aspirator and Other Test Suits
- Conclusion
Characteristics of Failures

● Finding 1
  ○ A majority (77%) of the failures require more than one input event to manifest
  ○ most of the failures (90%) require no more than 3
  ○ 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.

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

\{single event\} \{multiple events: 77%\}
Characteristics of Failures

● Finding 2
  ○ The specific order of events is important in 88% of the failures that require multiple input events.
Recall the structure of HDFS

A File is made of 64MB chunks

That are replicated for fault-tolerance

Chunks live on chunkservers

The master manages the file system namespace
1. upload blkA_100
   (100 is the generation stamp)

2. append to blkA

3. Start DN2
   namenode.register

NameNode

send to DN1
add blkA_100 to ‘needReplication’ queue

DataNode1
(only one DataNode started)

stores block:
blkA_100

‘needReplication’: blkA_100

check needReplication:
ask DN1 to replicate blkA_100 to DN2

‘gen-stamp not updated’!

updates to: blkA_101

gen-stamp mismatch:
blkA_100 (from NN) ≠ blkA_101 (local copy)

Refuse to replicate!
Characteristics of Failures

- 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.
  - It is not necessary to have a large cluster to test for and reproduce failure.
  - Does not contradict the conventional wisdom that distributed system failures are more likely to manifest on large clusters.

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

- Introduction
- Complexity of Failures And Opportunity for Improved Testing
- Role of Timing And Use of Logs in Reproducing Failures
- Catastrophic Failures
- Aspirator, A Simple Checker
- Comparisons: Aspirator and Other Test Suits
- Conclusion
The Role of Timing

What role does timing play in causing these failures?

- Complex timing => hard to reproduce error & debug
- Non-complex timing => can reproduce post-error & find fix!
Finding 4 - 74% of failures are deterministic

- **Deterministic** means that we are guaranteed to reproduce the error with the right sequence of input events
Finding 4 - 74% of failures are deterministic

The deterministic nature of most failures has positive consequences:

1. Need to explore combination/permutation of input events, but no need to worry about time gap between input events. **Implication:** can simulate events that only typically happen in long-running systems to reproduce errors

E.g ‘region split’ only occurs in HBase when region size gets too large, but we can just simulate & follow same sequence of input events to reproduce error

2. Failures reproducible after adding code to aid debugging e.g log output, tracing, activating debuggers
Finding 5 - Of 51 non-deterministic failures, 53% have timing constraints only on input events

- Error reproduction requires an input event to occur before/after other software execution event
- Since one part of timing dependency can be controlled by user, user can still carefully control timing of input events to reproduce error
Finding 5 - Of 51 non-deterministic failures, 53% have timing constraints only on input events

2 notes:
- Client write must occur **before** HMaster assigns region to new RegionServer
- Above operation not entirely controlled by user
Finding 5 - Of 51 non-deterministic failures, 53% have timing constraints only on input events

Remaining 24 non-deterministic errors due to shared-memory multi-threaded inter-leavings

- i.e these are concurrency bugs! One of atomicity violation, deadlock, lock contention

Harder to reproduce (and reason about)

- Adding a single logging statement can cause error to disappear!!
- Might need to manually add timing delays
Finding 5 - Of 51 non-deterministic failures, 53% have timing constraints only on input events.

```c
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.
The Role of Logs

What do logs tell us about the errors? Can they help us reproduce and ultimately fix errors?
Finding 6 - 76% of failures print explicit failure related error messages.

- Empirically, this rate is higher than that of a similar study on non-distributed systems (e.g Postgres, SVN) by author (43% of failures had explicit failure-related error messages logged)
- Potentially due to distributed systems developers paying more attention to logging due to difficulty of debugging and complexity of systems
Finding 7 - For a majority (84%) of failures, all of their triggering events are logged

- We can deterministically replay failures through existing log messages alone!
- Good news! But....

Figure 4: The logging comprehensiveness of the studied failures. Logging of both the input events and errors are considered. For failures requiring multiple events to trigger, we count it as “logged” only when all the required events are logged.
Finding 8 - Median no. of log messages printed by each failure is 824

- Logs are really noisy! => figuring out input events causing error is very tedious
- And above finding was only on minimal configuration required to reproduce error (i.e logs in production systems probably even noisier)
- Cannot just `grep` for logs w/ error messages, since input events that lead to error are often logged at INFO level (non-error logs)
- Space for future tools that can infer relevant error and input event messages from logs
Reproducing Failures

Conventional wisdom: reproducing error in large, production-grade distributed systems is hard

This reasoning is intuitive! Consider that:

- User’s input unavailable due to privacy protection
- Replicating an environment that mirrors the one in production is complex
- Cost of 3rd party libraries

Findings suggest otherwise!
Finding 9 - A majority of the production failures (77%) can be reproduced by a unit test.

Finding is counterintuitive to conventional wisdom, but a logical conclusion to findings

- Deterministic input event + Rich logging => Better reproducibility
- Even in non-deterministic events, in 53% of cases timing can be controlled in unit tests
- Turns out none of studied failures require specific values of user’s data
Finding 9 - A majority of the production failures (77%) can be reproduced by a unit test.

```java
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
}
```
Roadmap

- Introduction
- Complexity of Failures And Opportunity for Improved Testing
- Role of Timing & Logs in Reproducing Failures
- **Catastrophic Failures**
  - Aspirator, A Simple Checker
  - Comparisons: Aspirator and Other Test Suits
- Conclusion
Catastrophic Failures

What are catastrophic failures?
- Prevent all or majority of the users from their normal access to the system.

Why do we study them specifically?
- Have the largest business impact to the vendors.
What causes catastrophic failures?

- “Almost all catastrophic failures (92%) are result of incorrect handling of non-fatal errors explicitly signaled in software.”
Sample Trivial Mistakes (35%)

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.
Sample Trivial Mistakes (35%)
Breakdown of Catastrophic Failures

Figure 5: Break-down of all catastrophic failures by their error handling.
Sample System-specific Bugs (57%)

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.

System specific, but easily detectable (23%)
Sample System-specific Bugs (57%)

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.

Complex bugs (34%)
Breakdown of Catastrophic Failures

Figure 5: Break-down of all catastrophic failures by their error handling.
Insights

- Are outrages from large internet software vendors caused by incorrect error handling?
- How do node-failures lead to sever-level failures?
- How do we measure the applicability of the simple categories that are extracted from the bugs?
Roadmap

- Introduction
- Complexity of Failures And Opportunity for Improved Testing
- Role of Timing And Use of Logs in Reproducing Failures
- Catastrophic Failures
- **Aspirator, A Simple Checker**
- Comparisons: Aspirator and Other Test Suits
- Conclusion
A wild Cassandra has appeared.
Aspirator

Simple enough that Bili can understand it...

I choose you!
Aspirator - A simple, but effective solution

=========  
Aspirator: A simple checker for exception handler bugs  
=========  
- Aspirator is a tool that checks for trivial bug patterns in exception handlers for Java or JVM compatible programs. Specifically, it reports a warning if an important exception is ignored, system aborts on over-caught exceptions, or the exception handler contains "TODO" or "FIXME" in the comments. These trivial bugs in exception handlers have been shown to have caused a significant number of deadly failures for distributed systems. See more details in the paper:
Aspirator - How it works

1.) Scan Javabyte code, instruction by instruction

2.) Aspirator gives a warning if:
   a.) `catch` block is empty or contains “TODO” or “FIXME” in source code OR
   b.) `catch` block for a higher-level exception (Exception or Throwable) might catch multiple lower-level exceptions and calls `abort` or `exit` at the same time

3.) Aspirator doesn’t give a warning if
   a.) `try` block modifies a variable V AND
   b.) the value of V is checked in the basic block following the `catch` block

*Note* Developers can make exceptions for Aspirator to not give a warning for certain errors. Ex: Authors ignored all instances of `FileNotFoundException` exception.
Aspirator - example of no warnings

```java
uri = null;
try {
    uri = Util.fileAsURI(new File(uri));
} catch (IOException ex) {
    /* empty */
} if (uri == null) {
    // handle it here!
```
Aspirator - Reminder of Findings

- almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software.
- in 58% of the catastrophic failures, the underlying faults could easily have been detected through simple testing of error handling code.
# Aspirator - Results

<table>
<thead>
<tr>
<th>System</th>
<th>Handler blocks</th>
<th>Bug</th>
<th>Bad practice</th>
<th>False pos.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total / confirmed</td>
<td>ignore / abort / todo</td>
<td>total / confirmed</td>
<td>ignore / abort / todo</td>
</tr>
<tr>
<td>Cassandra</td>
<td>4,365</td>
<td>2 2</td>
<td>2 2</td>
<td>2 2</td>
</tr>
<tr>
<td>Cloudstack</td>
<td>6,786</td>
<td>27 24</td>
<td>25 - -</td>
<td>185 21</td>
</tr>
<tr>
<td>HDFS</td>
<td>2,652</td>
<td>24 9</td>
<td>23 - 1</td>
<td>32 5</td>
</tr>
<tr>
<td>HBase</td>
<td>4,995</td>
<td>16 16</td>
<td>11 3 2</td>
<td>43 6</td>
</tr>
<tr>
<td>Hive</td>
<td>9,948</td>
<td>25 15</td>
<td>23 - 2</td>
<td>54 14</td>
</tr>
<tr>
<td>Tomcat</td>
<td>5,257</td>
<td>7 4</td>
<td>6 1 -</td>
<td>23 3</td>
</tr>
<tr>
<td>Spark</td>
<td>396</td>
<td>2 2</td>
<td>- - 2</td>
<td>1 1</td>
</tr>
<tr>
<td>YARN/MR2</td>
<td>1,069</td>
<td>13 8</td>
<td>6 - 7</td>
<td>15 3</td>
</tr>
<tr>
<td>Zookeeper</td>
<td>1,277</td>
<td>5 5</td>
<td>5 - -</td>
<td>24 3</td>
</tr>
<tr>
<td>Total</td>
<td>36,745</td>
<td>121 85</td>
<td>101 4 16</td>
<td>379 58</td>
</tr>
</tbody>
</table>
Aspirator - Results

- Ran analysis on 9 distributed systems
- Each analysis finished within 15 seconds
- Used a MacBook Pro laptop with
  - 2.7Ghz Intel Core i7 processor
  - has memory footprints of less than 1.2GB
Aspirator Shortcomings

- limited to Java and languages with Java bytecode
- cannot determine how critical the bugs are
- warnings may not actually be bugs

However, developers can quickly assess the severity of the warnings
Roadmap

- Introduction
- Complexity of Failures And Opportunity for Improved Testing
- Role of Timing And Use of Logs in Reproducing Failures
- Catastrophic Failures
- Aspirator, A Simple Checker
- **Comparisons: Aspirator and Other Test Suits**
- Conclusion
What Makes Aspirator Special?

Cassandra, HBase, HDFS etc. all rigorously tested with many tools
What Makes Aspirator Special?

Cassandra, HBase, HDFS etc. all rigorously tested with many tools

- Unit Testing
- Random Error Injection
- Static Checking Tools
  - FindBugs is a popular program for Java applications that checks for potential errors based on more than 400 rules
What Makes Aspirator Special?

Cassandra, HBase, HDFS etc. all rigorously tested with many tools

- Unit Testing
- Random Error Injection
- Static Checking Tools
  - FindBugs is a popular program for Java applications that checks for potential errors based on more than 400 rules

How did Aspirator find 100s new bugs that these didn’t catch?
“Top-Down” Testing vs. “Bottom-Up” Testing

Top-Down

- What these other tools are based on
- What does it look like?
- Start the system with some test inputs
- Randomly inject errors along the way
- See what happens....
“Top-Down” Testing vs. “Bottom-Up” Testing

Top-Down

- What these other tools are based on
- What does it look like?
- Start the system with some test inputs
- Randomly inject errors along the way
- See what happens...

Less effective with large input size, lots of state
“Top-Down” Testing vs. “Bottom-Up” Testing

Bottom-Up

- Start with the error handling logic
- Focus on evaluating this error handling code
- Work backwards to find test cases to check this part of the program

Aspirator doesn’t exactly do this (no reverse-engineering test cases)

But it empirically validates the importance of thinking this way
Ennan used Aspirator! It's super effective!
CONCLUSIONS
92% of Catastrophic Errors <- Bad Error Handling
ARE YOU AN EXCEPTION?
BECAUSE I CAN'T WAIT TO CATCH YOU.
Prevention Methods

- Use tools similar to Aspirator
- Enforce code reviews on error-handling code.
- Look for paths that can actually reach and test error handling code.
  - Symbolic execution techniques.