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/

Presented by:
Casey Casalnuovo
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

<table>
<thead>
<tr>
<th>Software</th>
<th>Program language</th>
<th>Sampled failures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Java</td>
<td>40</td>
</tr>
<tr>
<td>HBase</td>
<td>Java</td>
<td>41</td>
</tr>
<tr>
<td>HDFS</td>
<td>Java</td>
<td>41</td>
</tr>
<tr>
<td>Hadoop MapReduce</td>
<td>Java</td>
<td>38</td>
</tr>
<tr>
<td>Redis</td>
<td>C</td>
<td>38</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>198</td>
</tr>
</tbody>
</table>

* Analysis of each failure can be found at: [http://www.eecg.toronto.edu/failureAnalysis/](http://www.eecg.toronto.edu/failureAnalysis/)
Sampling Details

- Blocker + critical errors considered.
- Selected issues only after 2010.
- Reporter and assignee not the same.
- Excluded duplicates (10 cases appeared in their sampling)
- From 17K total issues
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("…");

}
Finding I: *multiple* events are required

77% of the failures require more than one input events

**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

```java
/* 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
- R is assigned to slave C
- Slave C opens R

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

**Event 2:** Slave B dies

- R is assigned to slave C

/* Master: delete the
 * ZooKeeper node after
 * the region is opened */

```java
try {
    deleteZNode();
} catch (KeeperException e) {
    cluster.abort("...");
}
```
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
gen-stamp not updated!

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

Refuse to replicate!
Most errors required multiple events.

<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%\}

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

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

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%.
Number of nodes needed to reproduce the most errors is small.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>cumulative distribution function all failures</th>
<th>catastrophic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37%</td>
<td>43%</td>
</tr>
<tr>
<td>2</td>
<td>84%</td>
<td>86%</td>
</tr>
<tr>
<td>3</td>
<td>98%</td>
<td>98%</td>
</tr>
<tr>
<td>&gt; 3</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

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

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

Table 6: Number of failures that are deterministic.
Example of Nondeterministic Failure

- **HMaster**
  - Cannot connect to RS, thinks RS is dead
  - Recovers RS’ region data from its HLog
  - Recovery finishes
  - A new RS is assigned to serve the recovered region

- **RegionServer (RS)**
  1. RS network error
  2. RS is back online
  3. Rolls its HLog (HBase’s write-ahead-log)
  4. Receive client writes requests, log into the new HLog

*Order dependency btw. input operation and internal exe. state*

*Data writes are lost!*
# Reasons for Non-determinism

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

Table 7: Break-down of the non-deterministic failures. The “other” category is caused by nondeterministic behaviors from the OS and third party libraries.
Most failures (76%) and failure triggering events (84%) are logged.

Logs produce a median of 824 messages.
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
Unit-Testable Errors

<table>
<thead>
<tr>
<th>Software</th>
<th>% of failures reproducible by unit test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>73% (29/40)</td>
</tr>
<tr>
<td>HBase</td>
<td>85% (35/41)</td>
</tr>
<tr>
<td>HDFS</td>
<td>82% (34/41)</td>
</tr>
<tr>
<td>MapReduce</td>
<td>87% (33/38)</td>
</tr>
<tr>
<td>Redis</td>
<td>58% (22/38)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>77% (153/198)</td>
</tr>
</tbody>
</table>

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.
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]

Diagram:

- Faults
- Undetected Error (8%)
- Error detected, but wrongly handled (92%)
- Catastrophic failures (100%)
Trivial mistakes in error handling code

Example of abort in over-catch

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

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

Catastrophic failures (100%)

Faults

- Undetected Error (8%)
- Error detected, but wrongly handled (92%)
- Trivial mistakes (35%)
  - System specific, but easily detectable (23%)
  - Complex bugs (34%)
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

- Error detected, but wrongly handled (92%)
- Undetected Error (8%)
- Trivial mistakes (35%)
- Completely wrong
- System specific, but easily detectable (23%)
- Complex bugs (34%)

Catastrophic failures (100%)
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

/* Master: delete the
* ZooKeeper znode after
* the region is opened */

try {
    deleteZNode();
} catch (KeeperException e) {
    cluster.abort("..."神通);
}

**Event 2:** Slave B dies

R is assigned to slave C

Slave C opens R

Completely wrong
Additional Examples

Region (table) size grows > threshold

Split region
Remove old region’s metadata from META table
try {
    split(..);
} catch (Exception ex) {
    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.
Additional Examples

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

RemoteException is thrown due to glitch in namenode

Only intended for IncorrectVersionException

Figure 8: Entire HDFS cluster brought down by an over-catch.
Over half are trivial or easily detectable bugs

- Trivial mistakes (35%)
- System specific, but easily detectable (23%)
- Complex bugs (34%)
- Error detected, but wrongly handled (92%)
- Undetected Error (8%)
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/
False positive suppression

• Can avoid certain errors types:
  – FileNotFoundException was observed to contribute to the false positive rate, and was ignored.

• Find safety checks in the next basic block:

```java
uri = null;
try {
    uri = Util.fileAsURI(new File(uri));
} catch (IOException ex) { /* empty */ }
if (uri == null) { // handle it here! }
```
### Checking real-world systems

<table>
<thead>
<tr>
<th>System</th>
<th>Bugs</th>
<th>Bad practice</th>
<th>False positive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Training set</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>HBase</td>
<td>16</td>
<td>43</td>
<td>20</td>
</tr>
<tr>
<td>HDFS</td>
<td>24</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>Hadoop MapRed.2</td>
<td>13</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>Cloudstack</td>
<td>27</td>
<td>185</td>
<td>20</td>
</tr>
<tr>
<td>Hive</td>
<td>25</td>
<td>54</td>
<td>8</td>
</tr>
<tr>
<td><strong>Testing set</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tomcat</td>
<td>7</td>
<td>23</td>
<td>30</td>
</tr>
<tr>
<td>Spark</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Zookeeper</td>
<td>5</td>
<td>24</td>
<td>9</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>121</strong></td>
<td><strong>379</strong></td>
<td><strong>115</strong></td>
</tr>
</tbody>
</table>
New bugs can lead to catastrophic failures

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

- **Data loss**
  ```java
  try {
    journal.recover();
  } catch (IOException ex) {
  }
  ```

- **Cluster crash**
  - E.g., bugs found by “abort in over-catch” check

  *Cannot recover updates from journal*
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
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!