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/
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 weakeast 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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Catastrophic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>Java</td>
<td>40</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>Java</td>
<td>41</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>HDFS</td>
<td>Java</td>
<td>41</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Hadoop MapReduce</td>
<td>Java</td>
<td>38</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Redis</td>
<td>C</td>
<td>38</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>198</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

Not handled properly
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
- Slave C opens R

**Event 2:** Slave B dies

- R is assigned to slave C
- Slave C opens R

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

Only occur on long-running system (38%)
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
- Slave C opens R

**Event 2**: Slave B dies

- R is assigned to slave C

```java
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
- R is assigned to slave C
- Slave C opens R
- /* Master: delete the ZooKeeper node after the region is opened */

```java
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
```

```
Faults
```

```
Undetected Error
```

```
Catastrophic failures (100%)
```

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

Faults

Error detected, but wrongly handled (92%)

Undetected Error (8%)

Trivial mistakes (35%)

System specific, but easily detectable (23%)

Complex bugs (34%)

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

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

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

**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 */

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

Completely wrong
Over half are trivial or easily detectable bugs

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

Catastrophic failures (100%)

Faults
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 \( \frac{1}{3} \) 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

<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><strong>Total</strong></td>
<td><strong>121</strong></td>
<td><strong>379</strong></td>
<td><strong>115</strong></td>
</tr>
<tr>
<td><strong>Testing set</strong></td>
<td></td>
<td></td>
<td></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>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 in every system
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
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

/*
  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!