Understanding, Detecting, and Localizing Partial Failures in Large System Software [NSDI ‘20]

Chang Lou, Peng Huang, and Scott Smith, JHU

Presenter: Lilia Tang
Nov. 12, 2020
Motivation scenario

- Writes are timing out
- Reads still work, and creation times out
- Logs look good
- Resource usage metrics look good
Motivation scenario
Study methodology

- 100 partial failures in 5 widely-used systems
  - Zookeeper
  - Cassandra
  - HDFS
  - Apache
  - Mesos
Study methodology

**Partial Failure:** defined for a **process**

1. Fault happens
2. Fault does not crash process
3. Some functionality is slowed, or violates safety/liveness
Finding summary

- **Recency:** 54% are recent (last 3 years)
- **Root causes:** diverse root causes
- **Stuckness:** 48% cause functionality to be stuck
- **Zombies:** 13% have module which still executes after a severe error
- **Silence:** 15% are silent (data loss/corruption etc.)
- **Specific requirements:** 71% require specific environment, input, faults
- **Difficult recovery:** 68% require restart/repair
- **Long diagnosis time:** median diagnosis > 6 days
Diverse root causes

- 48% uncaught error, indefinite blocking, or buggy error handling

**Figure 2:** Root cause distribution. **UE:** uncaught error; **IB:** indefinite blocking; **EH:** buggy error handling; **DD:** deadlock; **PB:** performance bug; **LE:** logic error; **IL:** infinite loop; **RL:** resource leak.
Consequences

- **48%** cause some functionality to not make progress
- **17%** cause major slowdowns to the point of unusability
- **13%** zombies with undefined failure semantics

*Figure 3: Consequence of studied failures.*
Silent errors

- **15%** of errors are silent
  - Data loss/corruption, inconsistency, incorrect result
Specific triggering requirements

- 71% require specific environment, inputs, or fault events

```java
public byte[] readBuffer(String tag){
    int len = readInt(tag);
    if (len == -1) return null;
    byte[] arr = new byte[len];
    // more code
}
```
Long diagnosis time

- Median diagnosis time is > 6 days
  - Symptoms mislead debugging efforts
- Need more runtime info
  - Enable debug logging
  - Analyze the heap
  - Add instrumentation
Study influencing tool design

Specific triggering requirements → Detect at runtime

Long diagnosis time → Localize bugs

Uncaught errors → Construct detectors automatically

Stuck
Watchdog design goals

- Checkers **customized** to the component execution
  - Require specific inputs and bad state to manifest
  - **Solution:** mimic checkers
- Checkers need to be exercised on **synchronized state**
  - **Solution:** context hooks
- Checkers should **run concurrently** with the program
OmegaGen overview

- Generates mimic watchdogs
- Main idea: **program reduction**
  - Want to mimic behavior
  - But also help localize bugs
OmegaGen steps

1. Identify the long-running methods
2. Locate the vulnerable operations
3. Reduce the main program
4. Encapsulate the reduced program
5. Add checks to catch faults
Identify long-running methods

- Watchdogs only target checking continuously-executed code
- Traverse call graph
- If multiple call sites, add long-running predicate

```java
public class SyncRequestProcessor {
    public void run() {
        while (running) {
            if (logCount > (snapCount / 2)) {
                zks.takeSnapshot();
                ... reduce
            }
        }
    }
}
```

- Long-running loop
- No fixed iterations or over a collection
- Mark invoked methods
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null) {
            children = childs.toArray(new String[childs.size()]);
        }
    }
    path.append('/');
    int off = path.length();
    if (children != null) {
        for (String child : children) {
            path.delete(off, Integer.MAX_VALUE);
            path.append(child);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
What do we not include in the checker?

```java
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }

    path.append('/);
    int off = path.length();
    if (children != null) {
        for (String child : children) {
            path.delete(off, Integer.MAX_VALUE);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
```

Largely deterministic behavior
Can be tested with unit tests
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }
    path.append('/');
    int off = path.length();
    if (children != null) {
        for (String child : children) {
            path.delete(off, Integer.MAX_VALUE);
            path.append(child);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
Reduce the main program

- Keep vulnerable instructions
  - Keep one instance within the same method
- Reduced program retains call structure

```java
for (...) {
  ...
  oa.writeRecord(node, "node");
  ...
}
```

```java
<method1> (...) {
  ...
  oa.writeRecord(node, "node");
  ...
}
```

```java
<method2> (...) {
  ...
  oa.writeRecord(node, "node");
  ...
}
```

Keep over different method calls
Encapsulate reduced program and insert hooks

- Insert context hooks, passing the necessary arguments
- If context is ready and predicate is true, execute reduced method
- Skip if not

```java
public class DataTree {
    public void serializeNode(OutputArchive oa, ...) {
        ... String children[] = null;
        synchronized (node) {
            scount++;
            oa.writeRecord(node, "node");
            children = node.getChildren();
        }
        ... + ContextManger.serializeNode_reduced
            _args_setter(oa, node);
    }

    public class SyncRequestProcessor$Checker {
        public static void serializeNode_reduced(
            OutputArchive arg0, DataNode arg1) {
            arg0.writeRecord(arg1, "node");
        }

        public static void serializeNode_invoke() {
            Context ctx = ContextManger.
                serializeNode_reduced_context();
            if (ctx.status == READY) {
                factory
                    OutputArchive arg0 = ctx.args_getter(0);
                    DataNode arg1 = ctx.args_getter(1);
                serializeNode_reduced(arg0, arg1);
            }
        }
    }
```
Add checks to catch faults

- Focus on liveness and safety checks
- Liveness checks
  - **Solution**: fine-grained timeouts for local operations
- Safety checks
  - **Solution**: errors from vulnerable operations
  - Correctness violations are not a focus
    - Asserttion API for developers
Validate impact of faults

- Reported error can be transient or tolerable
- Transient errors
  - Solution: rerun the checker
- Tolerable errors
  - Solution: write short validation checks
Prevent side effects

- **Context replication**
  - Analyze all checker context
  - Replication setter to replicate when invoked
  - Lazy copying mechanism to when the getter needs it

- **Write redirection**
  - Redirect file/sockets to watchdog-specific ones

- **Read-operations**
  - Wait for the main program to finish reading, and get the value from the context
  - Implemented as a wrapper in the main program
## Evaluation scale

<table>
<thead>
<tr>
<th></th>
<th>ZooKeeper</th>
<th>Cassandra</th>
<th>HDFS</th>
<th>HBase</th>
<th>MapReduce</th>
<th>Yarn</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOC</td>
<td>28K</td>
<td>102K</td>
<td>219K</td>
<td>728K</td>
<td>191K</td>
<td>229K</td>
</tr>
<tr>
<td>Methods</td>
<td>3,562</td>
<td>12,919</td>
<td>79,584</td>
<td>179,821</td>
<td>16,633</td>
<td>10,432</td>
</tr>
<tr>
<td>Watchdogs</td>
<td>96</td>
<td>190</td>
<td>174</td>
<td>358</td>
<td>161</td>
<td>88</td>
</tr>
<tr>
<td>Methods</td>
<td>118</td>
<td>464</td>
<td>482</td>
<td>795</td>
<td>371</td>
<td>222</td>
</tr>
<tr>
<td>Operations</td>
<td>488</td>
<td>2,112</td>
<td>3,416</td>
<td>9,557</td>
<td>6,116</td>
<td>752</td>
</tr>
</tbody>
</table>
Checker coverage
Detection baselines

<table>
<thead>
<tr>
<th>Detector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client (Panorama [75])</td>
<td>instrument and monitor client responses</td>
</tr>
<tr>
<td>Probe (Falcon [82])</td>
<td>daemon thread in the process that periodically invokes internal functions with synthetic requests</td>
</tr>
<tr>
<td>Signal Resource</td>
<td>script that scans logs and checks JMX [40] metrics</td>
</tr>
<tr>
<td></td>
<td>daemon thread that monitors memory usage, disk and I/O health, and active thread count</td>
</tr>
</tbody>
</table>
Detection experiment

- Reproduced 22 real-world partial failures, mostly not from study
- Baseline and watchdogs run checks every second
- Measure the time needed to detect
- 20/22 cases detected with median 4.2 seconds
- Baselines only detected 14 combined

| Watch. | ZK1 | ZK2 | ZK3 | ZK4 | CS1 | CS2 | CS3 | CS4 | HF1 | HF2 | HF3 | HF4 | HB1 | HB2 | HB3 | HB4 | HB5 | MR1 | MR2 | MR3 | MR4 | YN1 |
|--------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Client |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Probe  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Signal |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Res.   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

Table 5: Detection times (in seconds) for the real-world cases in Table 10. ✗: undetected.
Localization accuracy

- Directly pinpoint 55% of the time
- 35% same function or call chain
- Baselines can only detect up to the faulty process
<table>
<thead>
<tr>
<th></th>
<th>ZK</th>
<th>CS</th>
<th>HF</th>
<th>HB</th>
<th>MR</th>
<th>YN</th>
</tr>
</thead>
<tbody>
<tr>
<td>watch</td>
<td>0–0.73</td>
<td>0–1.2</td>
<td>0</td>
<td>0–0.39</td>
<td>0</td>
<td>0–0.31</td>
</tr>
<tr>
<td>watch_v</td>
<td>0–0.01</td>
<td>0</td>
<td>0</td>
<td>0–0.07</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>probe</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>resource</td>
<td>0–3.4</td>
<td>0–6.3</td>
<td>0.05–3.5</td>
<td>0–3.72</td>
<td>0.33–0.67</td>
<td>0–6.1</td>
</tr>
<tr>
<td>signal</td>
<td>3.2–9.6</td>
<td>0</td>
<td>0–0.05</td>
<td>0–0.67</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7: False alarm ratios (%) of all detectors in the evaluated six systems. Each cell reports the ratio range under three setups (stable, loaded, tolerable). *watch_v*: watchdog with validators.
Performance and overhead

<table>
<thead>
<tr>
<th>Analysis</th>
<th>ZK</th>
<th>CS</th>
<th>HF</th>
<th>HB</th>
<th>MR</th>
<th>YN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation</td>
<td>21</td>
<td>166</td>
<td>75</td>
<td>92</td>
<td>55</td>
<td>50</td>
</tr>
<tr>
<td>ZK</td>
<td>43</td>
<td>103</td>
<td>130</td>
<td>953</td>
<td>131</td>
<td>89</td>
</tr>
</tbody>
</table>

**Table 8:** OmegaGen watchdog generation time (sec).

<table>
<thead>
<tr>
<th>Base</th>
<th>ZK</th>
<th>CS</th>
<th>HF</th>
<th>HB</th>
<th>MR</th>
<th>YN</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/ Watch.</td>
<td>399.8</td>
<td>3014.7</td>
<td>85.1</td>
<td>366.4</td>
<td>42.1</td>
<td>42.3</td>
</tr>
<tr>
<td>w/ Probe.</td>
<td>417.6</td>
<td>3128.2</td>
<td>89.4</td>
<td>374.3</td>
<td>44.9</td>
<td>44.9</td>
</tr>
<tr>
<td>w/ Resource.</td>
<td>424.8</td>
<td>3145.4</td>
<td>89.9</td>
<td>385.6</td>
<td>44.9</td>
<td>44.6</td>
</tr>
</tbody>
</table>

**Table 9:** System throughput (op/s) w/ different detectors.
Discovered new bug: ZOOKEEPER-3531

Synchronization on ACLCache cause cluster to hang when network/disk issues happen during datatree serialization

**Details**
- **Type:** Bug
- **Priority:** Critical
- **Affects Version/s:** 3.5.2, 3.5.3, 3.5.4, 3.5.5
- **Fix Version/s:** 3.6.0
- **Labels:** pull-request-available

**Description**
During our ZooKeeper fault injection testing, we observed that sometimes the ZK cluster could hang (requests time out, node status shows ok). After inspecting the issue, we believe this is caused by I/O (serializing ACLCache) inside a critical section. The bug is essentially similar to what is described in ZooKeeper-2201.

org.apache.zookeeper.server.DataTree#serialize calls the aclCache.serialize when doing datatree serialization, however, org.apache.zookeeper.server.ReferenceCountedACLCache#serialize could get stuck at OutputArchive.writeln due to potential network/disk issues. This can cause the system experiences hanging issues similar to ZooKeeper-2201 (any attempt to create/delete/modify the

**Dates**
- **Created:** 02/Sep/19 21:02