Understanding, Detecting and Localizing Partial Failures in Large System Software

Chang Lou*, Peng Huang, Scott Smith
All start from a puzzling incident..

ZooKeeper Cluster

Leader

Follower

Follower

On-call engineer
All start from a puzzling incident..

New Message: NEED HELP!

We met something puzzling. It has been going on for 10+ minutes.
All start from a puzzling incident..

READ OK
WRITE TIMEOUT
CREATE TIMEOUT

On-call engineer

logs

resource meter

Follower

Leader

Follower

“RUOK”

“IMOK”

Client
The failure off the radar

Client

VhKLe

(rTNNKNI)

{RLeep()

ReNFHeCrSDeCS()(

[5

RXNEhrONKzeF (NOFe)

VrKSe;SrKNI(pCSh" "pCSh)(

OTSpTS.

VrKSe:eEOrF(NOFe" "NOFe)(

[stuck at transient

network error

The failure off the radar

Follower

Leader Process

create('node1')

Request Processor

Serializer

Failure Detector

while (running)
{

sleep();

sendHeartbeat();

}

synchronized (node) {

output.writeString(path, "path");

output.writeRecord(node, "node");

}

stuck at transient network error

Leader Process

Client
Modern software is complex

- Protocol related workers
- I/O workers
- Background tasks
- Request workers
- Local database operators
- 233 live threads
Modern software is complex

233 live threads

Request workers

Local database operators

Protocol related workers

I/O workers

Background tasks

RequestResponseStage:6
RequestResponseStage:8
RequestResponseStage:5

MutationStage:36
MutationStage:41
MutationStage:42
MutationStage:44
MutationStage:46
MutationStage:43

GossipStage:1
AntiEntropyStage:1
MigrationStage:1
MiscStage:1
GossipTasks:1

WRITE:/10.142.0.6
WRITE:/10.142.0.6
WRITE:/10.142.0.4
WRITE:/10.142.0.4
WRITE:/10.142.0.5
WRITE:/10.142.0.5
WRITE:/10.142.0.3
Thread:6
WRITE:/10.142.0.3
New I/O worker #1
Thread:7
New I/O worker #2

OptionalTasks:1
MemTablePostFlusher:1
MemoryMeter:1
commitlog_archiver:1
COMMIT-LOG-ALLOCATOR
COMMIT-LOG-WRITER
PERIODIC-COMMIT-LOG-SYNCER
NonPeriodicTasks:1
pool-1-thread-1
Modern software is complex

Process can fail partially

Request workers
- RequestResponseStage:8
- RequestResponseStage:5
- MutationStage:36
- MutationStage:41
- MutationStage:42
- MutationStage:44
- MutationStage:46
- MutationStage:43

I/O workers
- WRITE/10.142.0.6
- WRITE/10.142.0.4
- WRITE/10.142.0.4
- WRITE/10.142.0.3
- WRITE/10.142.0.3
- New I/O worker #1
- Thread-7
- New I/O worker #2

Background tasks
- OptionalTasks:1
- MemtablePostFlusher:1
- MemoryMeter:1
- commitlog_archiver:1
- COMMIT-LOG-ALLOCATOR
- COMMIT-LOG-WRITER
- PERIODIC-COMMIT-LOG-SYNCER
- NonPeriodicTasks:1
- ppool-1-thread-1

Protocol related workers
- GossipStage:1
- AntiEntropyStage:1
- MigrationStage:1
- MiscStage:1
- GossipTasks:1

Local database operators

233 live threads

pid1

Number of Threads
- 200
- 250
- 300
- 350
- 400

Time Range: All
Real world outages caused by partial failures
Study methodology

- We study 100 partial failure cases from five large, widely-used software systems.

- Interestingly, 54% of them occur in the most recent three years’ software releases (average lifespan of all systems is 9 years).

<table>
<thead>
<tr>
<th>Software</th>
<th>Language</th>
<th>Cases</th>
<th>Versions</th>
<th>Date Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper</td>
<td>Java</td>
<td>20</td>
<td>17 (3.2.1–3.5.3)</td>
<td>12/01/2009–08/28/2018</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Java</td>
<td>20</td>
<td>19 (0.7.4–3.0.13)</td>
<td>04/22/2011–08/31/2017</td>
</tr>
<tr>
<td>HDFS</td>
<td>Java</td>
<td>20</td>
<td>14 (0.20.1–3.1.0)</td>
<td>10/29/2009–08/06/2018</td>
</tr>
<tr>
<td>Apache</td>
<td>C</td>
<td>20</td>
<td>16 (2.0.40–2.4.29)</td>
<td>08/02/2002–03/20/2018</td>
</tr>
<tr>
<td>Mesos</td>
<td>C++</td>
<td>20</td>
<td>11 (0.11.0–1.7.0)</td>
<td>04/08/2013–12/28/2018</td>
</tr>
</tbody>
</table>
A partial failure is, in a process $\pi$ to be when a fault does not crash $\pi$ but causes safety or liveness violation or severe slowness for some functionality $R_f \subsetneq R$. 

Study scope

Process-level

- PID1
- healthy/unhealthy

Service-level

- computing engine
- storage engine
- index instance

our focus

query APIs are broken

cluster

crash
Finding 1: root causes are diverse

- There is no single uniformed or dominating root cause.
- The top three (total 48%) root cause types are uncaught errors, indefinite blocking, and buggy error handling.

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.
Finding 2: nearly half cases cause stuck issues

- Nearly half (48%) of the partial failures cause some functionality to be stuck.

- 17% of the partial failures causes certain operations to take a long time to complete.
Finding 3: 15% cases are totally silent

- 15% of the partial failures are silent (including data loss, corruption, inconsistency, and wrong results).

Consequence of studied failures.

- Stuck
- Slow
- Zombie
- Omission
- Denial
- Corrupt
- Other

Diagram showing percent distribution of consequences with categories like Apache, Cassandra, Mesos, ZooKeeper, HDFS, and uncleaned buffer.
Finding 4: most cases are triggered by unique production workload or environment

- 71% of the partial failures are triggered by some specific environment condition, or special input in the production.

```java
public byte[] readBuffer(String tag){
    int len = readInt(tag);
    if (len == -1) return null;
    byte[] arr = new byte[len];
    // Read from the network packet
    // Compute schema_len from packet
    schema_len = 0x6edd0b51;  // 1859980113
}
```

```
0200 6b 2d 00 00 00 09 31 32 37 2e 30 2e 30 2e 31 00 k-....127.0.0.1.
0210 7c 0e 5b 86 df f3 fc 6e dd 0b 51 |.
0220 00 00 00 00 61 6e 79 6f 6e 65 |[....n..Q.....
0220 00 00 00 00 06 61 6e 79 6f 6e 65 ....anyone....
```

ZK-602 ZK-914
Finding 5: debugging time is long

- The median diagnosis time is 6 days and 5 hour.
  - confusing symptoms of the failures mislead the diagnosis direction
  - insufficient exposure of runtime information in the faulty process

- enable DEBUG log
- analyze heap
- instrument code
How to deal with partial failures

Static Approach

Dynamic Approach

WE NEED MORE POWERFUL RUNTIME CHECKERS

no unique production env/workload

existing detectors are too shallow
Manual vs generated checkers

- Ask developers to manually add defensive checks?

```java
datagramSocket.receive();
...  
outputArchive.writeRecord(record);
...
randomAccessFile.seek();
...
CalcUtil.hash(buffer);
...
allocateMemory(size);
...
```

```java
+ long start = System.nanoTime();
+ long elapsedTime = System.nanoTime() - start;
+ try{
+ } catch (IOException e) {...}
+ try{
+ } catch (NullPointerException e) {...}
+ try{
+ } catch (OutOfMemoryError e) {...}
+ } catch (OutOfMemoryError e) {...}
```
Manual vs generated checkers

- Systematically generate checkers to ease developers’ burden
  - challenge: difficult to automate for all cases
  - opportunity: most of partial failures do not rely on deep semantic understanding to detect, such checkers can potentially be automatically constructed
Design principle: checkers intersect with the execution of a monitored process

existing approach

main program
checker

our approach

main program
checker
Intrinsic watchdog: Runtime

- Request Listener
- Snapshot Manager
- Replication Engine
- Compaction Worker
- Session Manager

- Contexts
- watchdog
- checkers
- driver

main program
Characteristic I: customized

- Request Listener
- Snapshot Manager
- Replication Engine
- Compaction Worker
- Session Manager
- Contexts
- main program
- checkers
- driver
Characteristic II: stateful

- Request Listener
- Snapshot Manager
- Replication Engine
- Compaction Worker
- Session Manager
- Contexts
- watchdog hooks
- main program
- checkers
- driver
Characteristic III: concurrent

- Request Listener
- Snapshot Manager
- Replication Engine
- Compaction Worker
- Session Manager

main program

Contexts

checkers

driver
Core idea: mimic checking

Client

create('node1')

Leader Process

Request Processor

Serializer

Mimic checker

synchronized (node) {
    output.writeString(path, "path");
    output.writeRecord(node, "node");
}

stuck at transient network error

output.writeString(path, "path");
Tool overview

- **OmegaGen**
  - a prototype that systematically generate mimic-type watchdogs for system softwares

- **core technique:** program reduction

```bash
% ./omegagen -jar zookeeper-3.4.6.jar -m zookeeper.manifest
analyzing..
generating..
instrumenting..
repackaging..
done. Total 1min 6s.
% ls output/
zookeeper-3.4.6-with-wd.jar
```
What is program reduction?

- Given a program $P$, create a watchdog $W$ that can detect partial failures in $P$ without imposing on $P$'s execution.
1) We should not put everything into 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);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
```
1) We should not put everything into 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);
        }
    }
}
```
2) We need not put everything into 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> children = node.getChildren();
        if (children != null)
            children = children.toArray(new String[children.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);
        }
    }
}
```
2) We need not put everything into 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);
        }
    }
}
```

- convert string
- convert array
- append path
- iterate children and modify path

A lot of operations are logically deterministic and should be checked before production.
2) We need not put everything into 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);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
```

- do I/O + in synchronized block

Some operations are more vulnerable in the production environment

We need not put everything into checker
do I/O + in synchronized block

some operations are more vulnerable in the production environment
Five steps

- #1 locate long-running regions
- #2 reduce the program
- #3 locate vulnerable operations
- #4 encapsulate watchdog checkers
- #5 insert watchdog hooks
Step #1 locate long-running regions

**Initialization stage**

public class SyncRequestProcessor {
  public void run() {
    int logCount = 0;
    setRandRoll(r.nextInt(snapCount/2));

    while (running) {
      ...
      if (logCount > (snapCount / 2))
        zks.takeSnapshot();
    }

    LOG.info("SyncRequestProcessor exited!");
  }
}

**Long-running stage**

**Cleanup stage**
Step#2 reduce the program

```java
public class SyncRequestProcessor {
    public static void serializeSnapshot(DataTree dt, ...) {
        dt.serialize(oa, "tree");
    }
}

public class DataTree{
    public void serialize(OutputArchive oa, String tag) {
        scout = 0;
        serializeNode(oa, new StringBuilder(""));
        ... keep reducing
    }
}
```
Step#3 locate vulnerable operations

```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();
    ...
}
```

our heuristic

I/O, synchronization, resource, communication related method invocations, ...

Step#3 locate vulnerable operations

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();
    ...
}
Step#4 encapsulate watchdog checkers

```java
public class SyncRequestProcessor$Checker {
    public static void serializeNode_reduced(OutputArchive arg0, DataNode arg1) {
        try{
            arg0.writeRecord(arg1, "node");
        } catch (Throwable ex) {
            ...
        }
    }

    public static Status checkTargetFunction0() {
        Context ctx = ContextFactory.serializeNode_reduced_context();
        if (ctx.status == READY) {
            OutputArchive arg0 = ctx.args_getter(0);
            DataNode arg1 = ctx.args_getter(1);
            executor.runAsyncWithTimeout(serializeSnapshot_reduced(arg0, arg1), TIMEOUT);
        } else {
            LOG.debug("checker context not ready");
        }
    }
}
```
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();
    ...
}
Validation

- An error reported by a watchdog checker could be transient or tolerable.
  - e.g. a transient network delay that caused no damage

- Watchdog driver by default simply re-executes the checker and compare for transient errors.
  - OmegaGen also allows developers write their own user-defined validation tasks to check some entry functions, e.g., `processRequest(req)`
  - The tool would automate the part of deciding which validation task to invoke depending on which checker failed.
Prevent Side Effects

- **Context Replication (memory isolation)**
  - context manager will replicate the checker context so that any modifications are contained in the watchdog’s state

![Diagram showing the replication of DataNodes and their states over time.](image-url)
Prevent Side Effects

- Context Replication (memory isolation)
  - to reduce performance overhead: immutability analysis + lazy copy
Prevent Side Effects

- **Context Replication (memory isolation)**
  - check consistency before copying and invocation with hashCode and versioning

```
main program
```

```
DataNode 1
```

```
DataNode 2
```

```
DataNode 2'
```

```
DataNode 3
```

```
DataNode 3'
```

```
DataNode3 (m)
```

```
unmatch, skip!
```

```
check
```

```
hash(val1) != hash(val2)
```

```
check
```

```
lazy copy
```

```
time
```

Prevent Side Effects

- **I/O Redirection and Idempotent Wrappers (I/O isolation)**
  - write: file-related resource replicated with target path changed to test file
  - read: watchdogs pre-read contexts and cache

### BufferStream

- **buffer**
- **flags**
- **outputFile (a.out)**

### Diagram

- **write-redirection**
  - copy: buffer → buffer'
  - copy: flags → flags'
  - copy: outputFile (a.out) → outputFile (a.out.mirror)

- **read-redirection**
  - **wrapper**
    - check
  - **time**
    - P
    - timeout
    - try to fetch
    - hang
  - **check**
Evaluation
## Evaluated systems

<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>
</tbody>
</table>

**Scale of evaluated system software**
## Watchdog generation

<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><strong>Watchdogs</strong></td>
<td>96</td>
<td>190</td>
<td>174</td>
<td>358</td>
<td>161</td>
<td>88</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td>118</td>
<td>464</td>
<td>482</td>
<td>795</td>
<td>371</td>
<td>222</td>
</tr>
<tr>
<td><strong>Operations</strong></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>

**Number of watchdogs and checkers generated**
22 real-world partial failures reproduced for evaluation

<table>
<thead>
<tr>
<th>JIRA Id.</th>
<th>Id.</th>
<th>Root Cause</th>
<th>Conseq.</th>
<th>Sticky?</th>
<th>Study?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper-2201</td>
<td>ZK1</td>
<td>Bad Synch.</td>
<td>Stuck</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ZooKeeper-602</td>
<td>ZK2</td>
<td>Uncaught Error</td>
<td>Zombie</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ZooKeeper-2325</td>
<td>ZK3</td>
<td>Logic Error</td>
<td>Inconsist</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ZooKeeper-3131</td>
<td>ZK4</td>
<td>Resource Leak</td>
<td>Slow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cassandra-6364</td>
<td>CS1</td>
<td>Uncaught Error</td>
<td>Zombie</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Cassandra-6415</td>
<td>CS2</td>
<td>Indefinite Blocking</td>
<td>Stuck</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cassandra-9549</td>
<td>CS3</td>
<td>Resource Leak</td>
<td>Slow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cassandra-9486</td>
<td>CS4</td>
<td>Performance Bug</td>
<td>Slow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HDFS-8429</td>
<td>HF1</td>
<td>Uncaught Error</td>
<td>Stuck</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HDFS-11377</td>
<td>HF2</td>
<td>Indefinite Blocking</td>
<td>Stuck</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>HDFS-11352</td>
<td>HF3</td>
<td>Deadlock</td>
<td>Stuck</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HDFS-4233</td>
<td>HF4</td>
<td>Uncaught Error</td>
<td>Data Loss</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HBase-18137</td>
<td>HB1</td>
<td>Infinite Loop</td>
<td>Stuck</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HBase-16429</td>
<td>HB2</td>
<td>Deadlock</td>
<td>Stuck</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HBase-21464</td>
<td>HB3</td>
<td>Logic Error</td>
<td>Stuck</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HBase-21357</td>
<td>HB4</td>
<td>Uncaught Error</td>
<td>Denial</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HBase-16081</td>
<td>HB5</td>
<td>Indefinite Blocking</td>
<td>Silent</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MapReduce-6351</td>
<td>MR1</td>
<td>Deadlock</td>
<td>Stuck</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MapReduce-6190</td>
<td>MR2</td>
<td>Infinite Loop</td>
<td>Stuck</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MapReduce-6957</td>
<td>MR3</td>
<td>Improper Err Handling</td>
<td>Stuck</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MapReduce-3634</td>
<td>MR4</td>
<td>Uncaught Error</td>
<td>Zombie</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yarn-4254</td>
<td>YN1</td>
<td>Improper Err Handling</td>
<td>Stuck</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
# Detection Setup

<table>
<thead>
<tr>
<th>Detector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client (Panorama [OSDI '18])</td>
<td>instrument and monitor client responses</td>
</tr>
<tr>
<td>Probe (Falcon [SOSP '11])</td>
<td>daemon thread in the process that periodically invokes internal functions with synthetic requests</td>
</tr>
<tr>
<td>Signal</td>
<td>script that scans logs and checks JMX metrics</td>
</tr>
<tr>
<td>Resource</td>
<td>daemon thread that monitors memory usage, disk and I/O health, and active thread count</td>
</tr>
</tbody>
</table>

**Baseline checkers**
## Detection Time

<table>
<thead>
<tr>
<th></th>
<th>ZK1</th>
<th>ZK2</th>
<th>ZK3</th>
<th>ZK4</th>
<th>CS1</th>
<th>CS2</th>
<th>CS3</th>
<th>CS4</th>
<th>HF1</th>
<th>HF2</th>
<th>HF3</th>
<th>HF4</th>
<th>HB1</th>
<th>HB2</th>
<th>HB3</th>
<th>HB4</th>
<th>HB5</th>
<th>MR1</th>
<th>MR2</th>
<th>MR3</th>
<th>MR4</th>
<th>YN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>×</td>
<td>2.47</td>
<td>2.27</td>
<td>×</td>
<td>441</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>4.81</td>
<td>×</td>
<td>6.62</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>8.54</td>
<td>7.38</td>
</tr>
<tr>
<td>Probe</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>15.84</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>4.71</td>
<td>×</td>
<td>7.76</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Signal</td>
<td>12.2</td>
<td>0.63</td>
<td>1.59</td>
<td>0.4</td>
<td>5.31</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>0.77</td>
<td>0.619</td>
<td>×</td>
<td>0.62</td>
<td>61.0</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>0.60</td>
</tr>
<tr>
<td>Res.</td>
<td>5.33</td>
<td>0.56</td>
<td>0.72</td>
<td>17.17</td>
<td>209.5</td>
<td>×</td>
<td>-19.65</td>
<td>×</td>
<td>-3.13</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>0.83</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>0.60</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Watch</td>
<td>4.28</td>
<td>-5.89</td>
<td>3.00</td>
<td>41.19</td>
<td>-3.73</td>
<td>4.63</td>
<td>46.56</td>
<td>38.72</td>
<td>1.10</td>
<td>6.20</td>
<td>3.17</td>
<td>2.11</td>
<td>5.41</td>
<td>7.89</td>
<td>×</td>
<td>0.80</td>
<td>5.89</td>
<td>1.01</td>
<td>4.07</td>
<td>1.46</td>
<td>4.68</td>
<td>×</td>
</tr>
</tbody>
</table>

Detection times (in secs) for the real-world cases
Detection Localization

void serializeNode()
{
    size = size + 1;
    oa.writeInt(size);
    oa.flush();
}

void serializeNode()
{
    size = size + 1;
    oa.writeInt(size);
    oa.flush();
}

save()
  ...serializeSnapshot()
  save()
  ...serialize()
  ....serialize()
  ....serialize()
  ....serialize()
  ....serialize()
## Detection Localization

<table>
<thead>
<tr>
<th>ZK1</th>
<th>ZK2</th>
<th>ZK3</th>
<th>ZK4</th>
<th>CS1</th>
<th>CS2</th>
<th>CS3</th>
<th>CS4</th>
<th>HF1</th>
<th>HF2</th>
<th>HF3</th>
<th>HF4</th>
<th>HB1</th>
<th>HB2</th>
<th>HB3</th>
<th>HB4</th>
<th>HB5</th>
<th>MR1</th>
<th>MR2</th>
<th>MR3</th>
<th>MR4</th>
<th>YN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>n/a</td>
<td>●</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>○</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Probe</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Signal</td>
<td>●</td>
<td>❆</td>
<td>●</td>
<td>❆</td>
<td>●</td>
<td>❆</td>
<td>●</td>
<td>❆</td>
<td>●</td>
<td>❆</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Res.</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>n/a</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Watch</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>●</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>●</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>●</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
<td>❆</td>
</tr>
</tbody>
</table>

### Failure localization for the real-world cases.

```c
void serializeNode()
{
    size = size + 1;
    oa.writeInt(size);
    oa.flush();
}
```

```c
void serializeNode()
{
    size = size + 1;
    oa.writeInt(size);
    oa.flush();
}
```

```c
save()
    .serializeSnapshot()
    ....serialize()
    ....serializeNode()
```

```c
SerializerThread@run()
...
save()
    .serializeSnapshot()
    ....serialize()
    ....serializeNode()
```

- **faulty instr**
- **faulty func**
- **faulty call chain**
- **faulty entry**
- **faulty proc**
- n/a **no localize**
- ○ **misjudged proc**

### Example:
- pid1
- NOT DETECTED
- pid1
- pid2
Discover new bug

```java
public void serializeAcls()
{
    ...  
    synchronized (this)
    {
        oa.writeInt(longKeyMap.size(), "map");
        for (Map.Entry<Long, List<ACL>> val : longKeyMap.entrySet()) {
            oa.writeLong(val.getKey(), "long");
        }
    }
}
```
Discover new bug

ZooKeeper / ZOOKEEPER-3531

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

Details
- Type: Bug
- Status: RESOLVED
- Priority: Critical
- Resolution: Fixed
- Affects Version/s: 3.5.2, 3.5.3, 3.5.4, 3.5.5
- Fix Version/s: 3.6.0
- Component/s: None
- 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.writeInt 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

People
- Assignee: Chang Lou
- Reporter: Chang Lou
- Votes: 0

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

## False alarms

<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>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>
<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>
</tbody>
</table>

False alarm ratios of all detectors for six systems under various setups
System overhead

<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>Base</td>
<td>428.0</td>
<td>3174.9</td>
<td>90.6</td>
<td>387.1</td>
<td>45.0</td>
<td>45.0</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>
<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>
</tbody>
</table>

Throughput (op/s) w/ different checkers

5.0%-6.6% throughput overhead w/ watchdog

Heap memory usage w/ and w/o watchdogs

4.3% (avg) memory overhead w/ watchdog
Conclusions

- Modern software are increasingly complex and often fail partially.
  - these partial failures cannot be detected by process-level failure detectors.
- We conducted a study on 100 partial failure cases.
- OmegaGen: a static analysis tool that automatically generates customized checkers.
  - successfully generate checkers for six systems and checkers can detect & localize 18/22 real-world partial failures.
  - watchdog report helps to quickly discover a new bug in the latest zookeeper.
Related Work

- **Partial failures**
  - Fail–Stutter [HotOS ’01], IRON [SOSP’05], Limplock [SoCC ’13], Fail–Slow Hardware [FAST ’18], Gray Failure [HotOS ’17]

- **Failure detection**
  - Gossip [Middleware ’98], ϕ [SRDS ’04], Falcon [SOSP ’11], Pigeon [NSDI ’13], Panorama [OSDI’18]

- **Software invariant generation**
  - Daikon [ICSE’99], InvGen [CAV’09], PCHECK [OSDI’16]