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

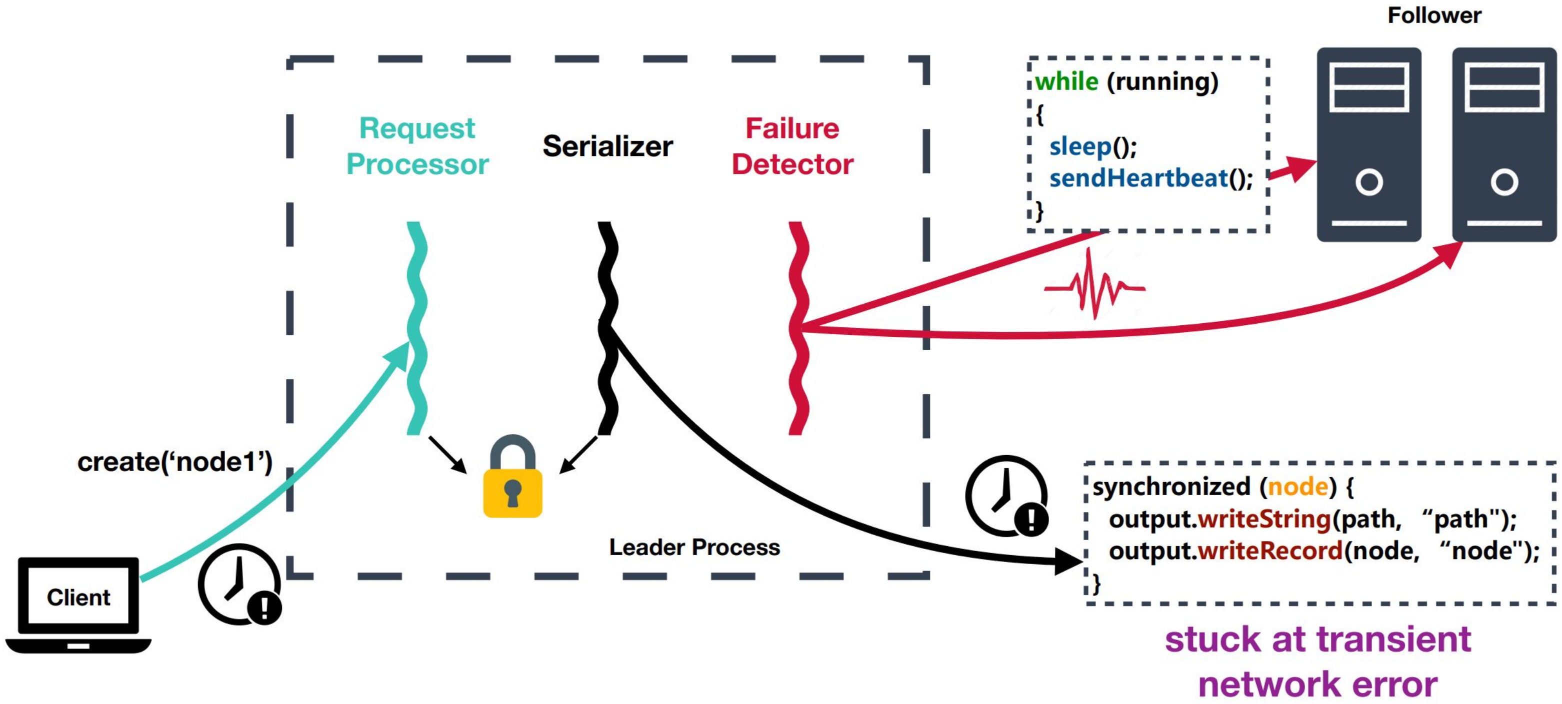
Chang Lou, Peng Huang, and Scott Smith, *JHU*

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

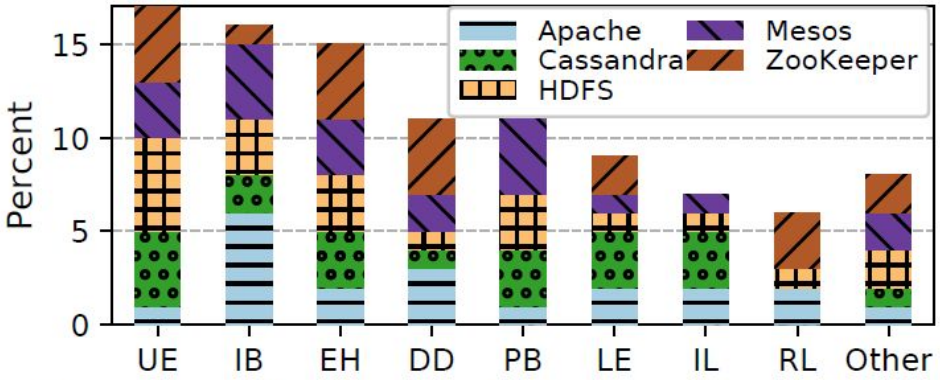


Figure 2: Root cause distribution. *UE*: uncaught error; *IB*: indefinite blocking; *EH*: buggy error handling; *DD*: deadlock; *PB*: performance bug; *LE*: logic error; *IL*: infinite loop; *RL*: resource leak.

Consequences

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

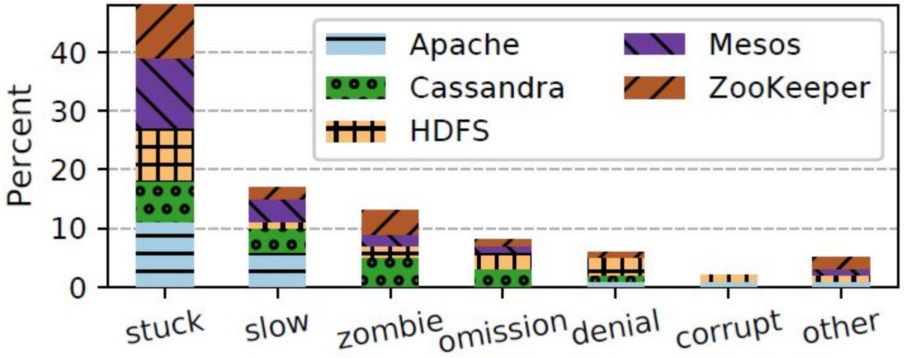
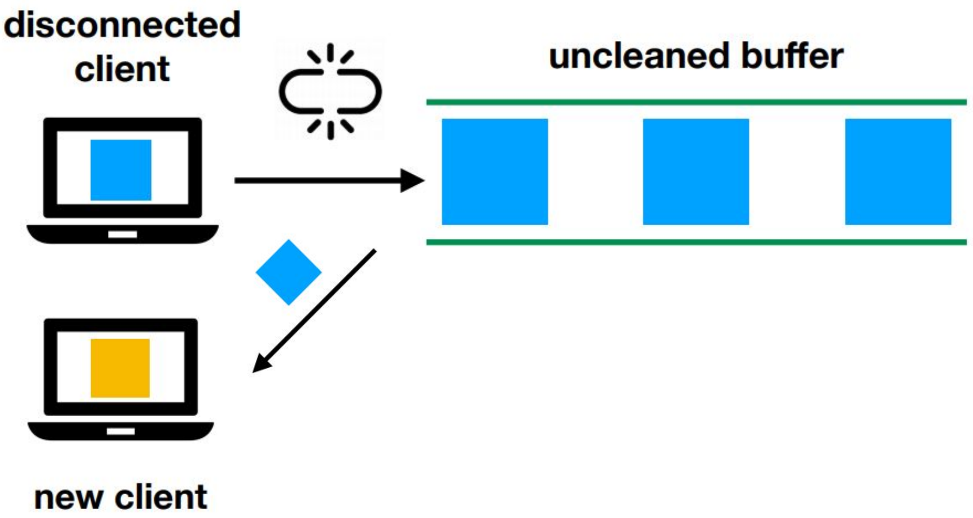


Figure 3: Consequence of studied failures.

Silent errors

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



Specific triggering requirements

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

```
public byte[] readBuffer(String tag){
    int len = readInt(tag);
    if (len == -1) return null;
    byte[] arr = new byte[len];
```

schema_len

0x6edd0b51=1859980113

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 Stuck	→	Construct detectors automatically

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

❶ identify long-running region

❸ reduce

Long-running loop
No fixed iterations or over a collection

Mark invoked methods

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

What do we not include in the checker?

```
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

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

Heuristic: file I/O
Sync, resource allocation, event polling, async waiting, invocation with external input, I/O

Reduce the main program

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

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

```
oa.writeRecord(node, "node");
```

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

```
<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);
}
```

3 reduce

2 locate vulnerable operations

4 insert context hooks

```java
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) {
      OutputArchive arg0 = ctx.args_getter(0);
      DataNode arg1 = ctx.args_getter(1);
      serializeNode_reduced(arg0, arg1);
    }
  }
```

4 generate context factory

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
    - Assertion 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

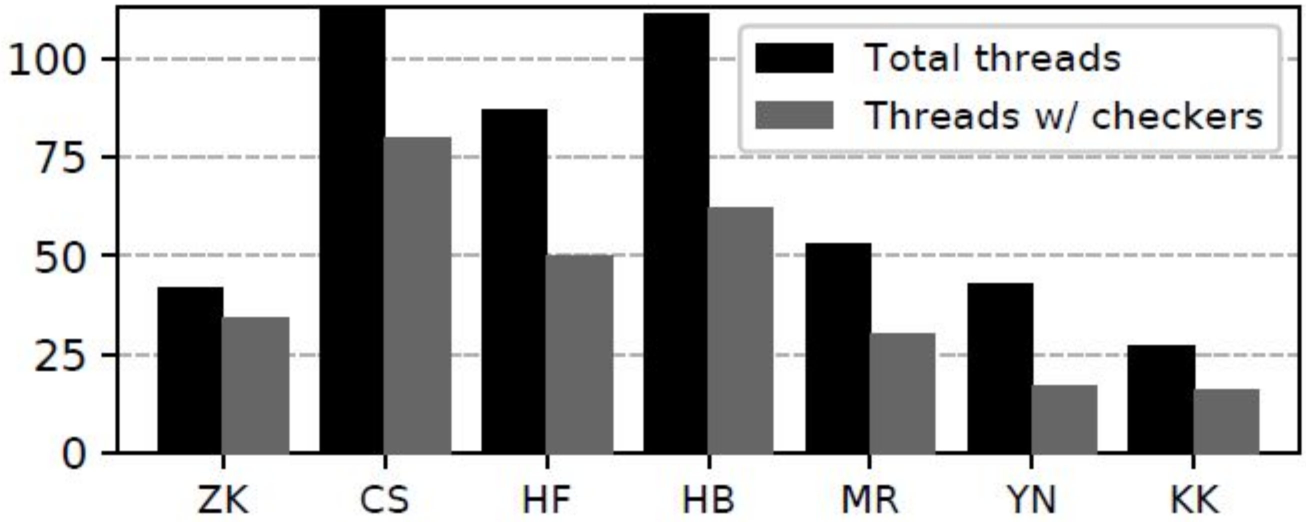
- Context replication
  - Analyze all checker context
  - Replication setter to replicate when invoked
  - Lazy copying mechanism to when the getter needs it
- Write redirection
  - Redirect file/sockets to watchdog-specific ones
- Read-operations
  - Wait for the main program to finish reading, and get the value from the context
  - Implemented as a wrapper in the main program

Evaluation scale

	ZooKeeper	Cassandra	HDFS	HBase	MapReduce	Yarn
SLOC	28K	102K	219K	728K	191K	229K
Methods	3,562	12,919	79,584	179,821	16,633	10,432

	ZooKeeper	Cassandra	HDFS	HBase	MapReduce	Yarn
Watchdogs	96	190	174	358	161	88
Methods	118	464	482	795	371	222
Operations	488	2,112	3,416	9,557	6,116	752

Checker coverage



Detection baselines

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

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

	ZK1	ZK2	ZK3	ZK4	CS1	CS2	CS3	CS4	HF1	HF2	HF3	HF4	HB1	HB2	HB3	HB4	HB5	MR1	MR2	MR3	MR4	YN1
Watch.	4.28	-5.89	3.00	41.19	-3.73	4.63	46.56	38.72	1.10	6.20	3.17	2.11	5.41	7.89	✖	0.80	5.89	1.01	4.07	1.46	4.68	✖
Client	✖	2.47	2.27	✖	441	✖	✖	✖	✖	✖	✖	✖	✖	4.81	✖	6.62	✖	✖	✖	✖	8.54	7.38
Probe	✖	✖	✖	✖	15.84	✖	✖	✖	✖	✖	✖	✖	✖	4.71	✖	7.76	✖	✖	✖	✖	✖	✖
Signal	12.2	0.63	1.59	0.4	5.31	✖	✖	✖	✖	✖	✖	0.77	0.619	✖	0.62	61.0	✖	✖	✖	✖	0.60	1.16
Res.	5.33	0.56	0.72	17.17	209.5	✖	-19.65	✖	-3.13	✖	✖	0.83	✖	✖	✖	0.60	✖	✖	✖	✖	✖	✖

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

	ZK	CS	HF	HB	MR	YN
watch.	0–0.73	0–1.2	0	0–0.39	0	0–0.31
watch_v.	0–0.01	0	0	0–0.07	0	0
probe	0	0	0	0	0	0
resource	0–3.4	0–6.3	0.05–3.5	0–3.72	0.33–0.67	0–6.1
signal	3.2–9.6	0	0–0.05	0–0.67	0	0

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

	ZK	CS	HF	HB	MR	YN
Analysis	21	166	75	92	55	50
Generation	43	103	130	953	131	89

Table 8: OmegaGen watchdog generation time (sec).

	ZK	CS	HF	HB	MR	YN
Base	428.0	3174.9	90.6	387.1	45.0	45.0
w/ Watch.	399.8	3014.7	85.1	366.4	42.1	42.3
w/ Probe.	417.6	3128.2	89.4	374.3	44.9	44.9
w/ Resource.	424.8	3145.4	89.9	385.6	44.9	44.6

Table 9: System throughput (op/s) w/ different detectors.

Discovered new bug: ZOOKEEPER-3531

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 dataree serialization, however, org.apache.zookeeper.server.ReferenceCountedACLCache#serialize could get stuck at OutputArchieve.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 Vote for this issue

Watchers:

5 Start watching this issue

Dates

Created:

02/Sep/19 21:02