

# Understanding, Detecting and Localizing Partial Failures in Large System Software<sup>1</sup>

November 6, 2020

---

<sup>1</sup>Chang Lou, Peng Huang, and Scott Smith. "Understanding, Detecting and Localizing Partial Failures in Large System Software". In: *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 2020, pp. 559–574.

# Overview

Problem definition

Case Study  
Findings

Motivation

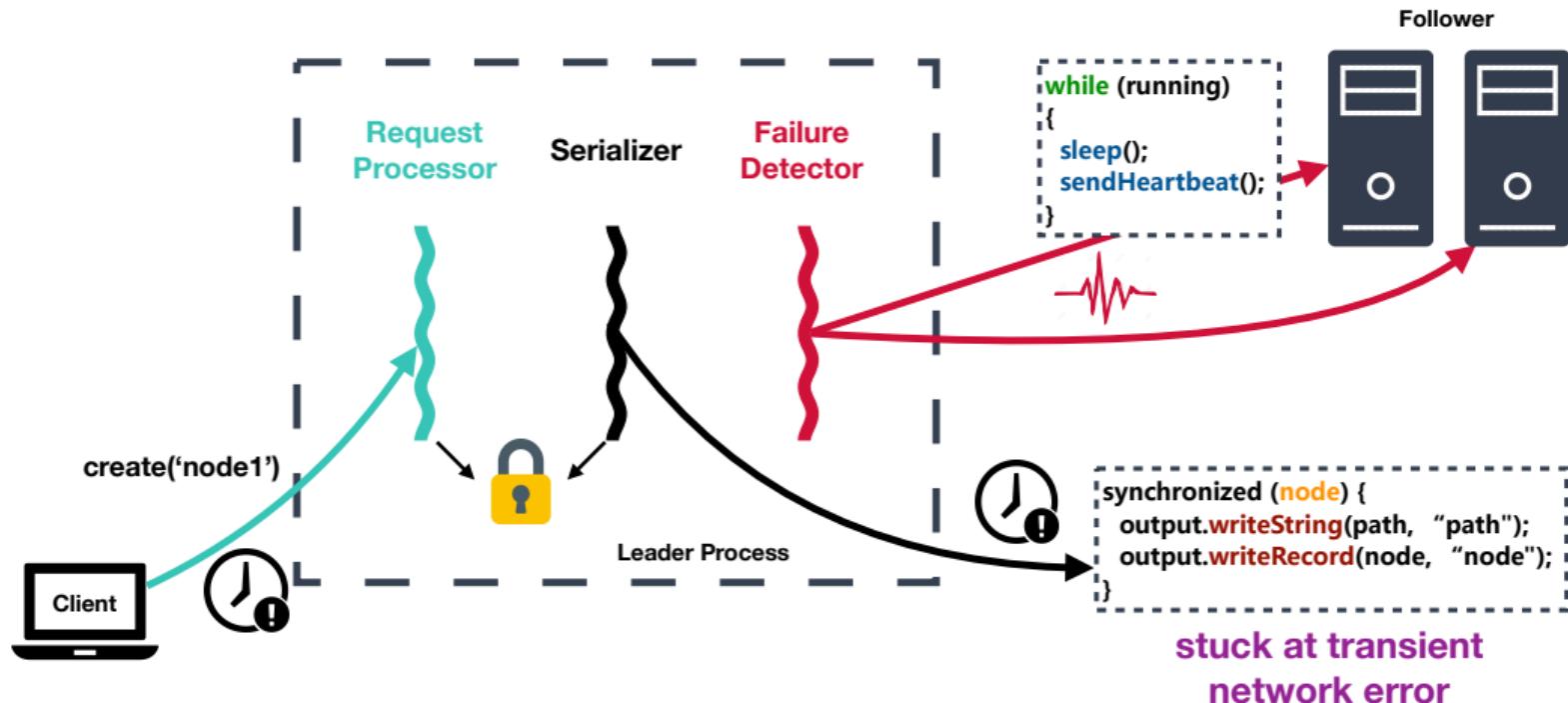
Proposed Design  
Ideas  
Implementation

Evaluation

Summary

# What is a Partial Failure?

An Example

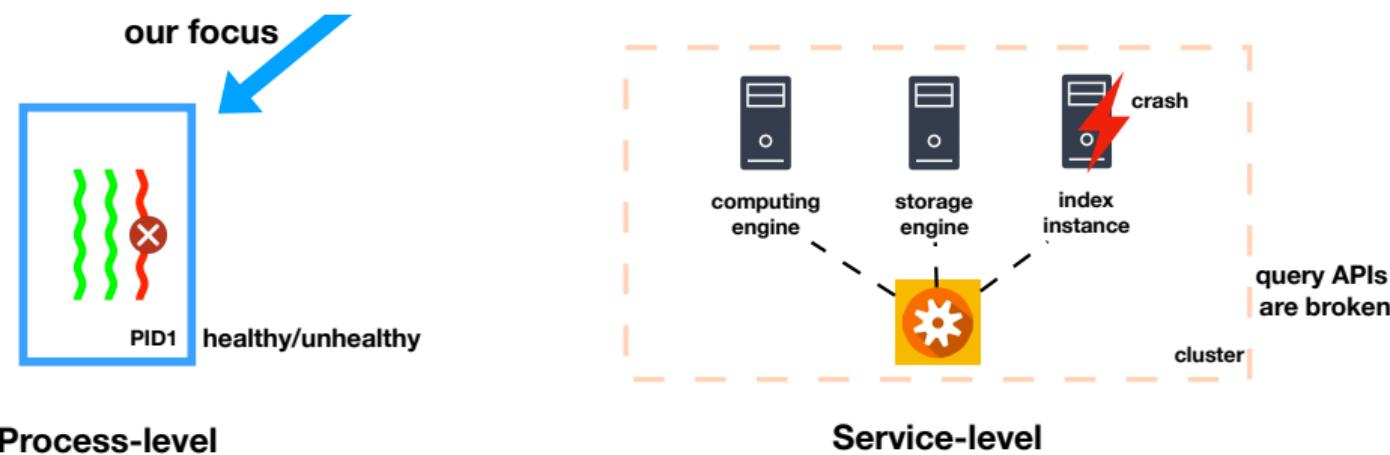


# What is a Partial Failure?

## Definition

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  $R_f \subsetneq R$

**Scope:** In this paper, we will specify the partial failure at the **process** granularity instead of **service**.



# Study methodology

## 100 partial failure cases from five large, widely-used software systems

- ▶ Crawl all bug tickets tagged with critical priorities in the official bug trackers
- ▶ Filter tickets from testing and randomly sample the remaining failures tickets.

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

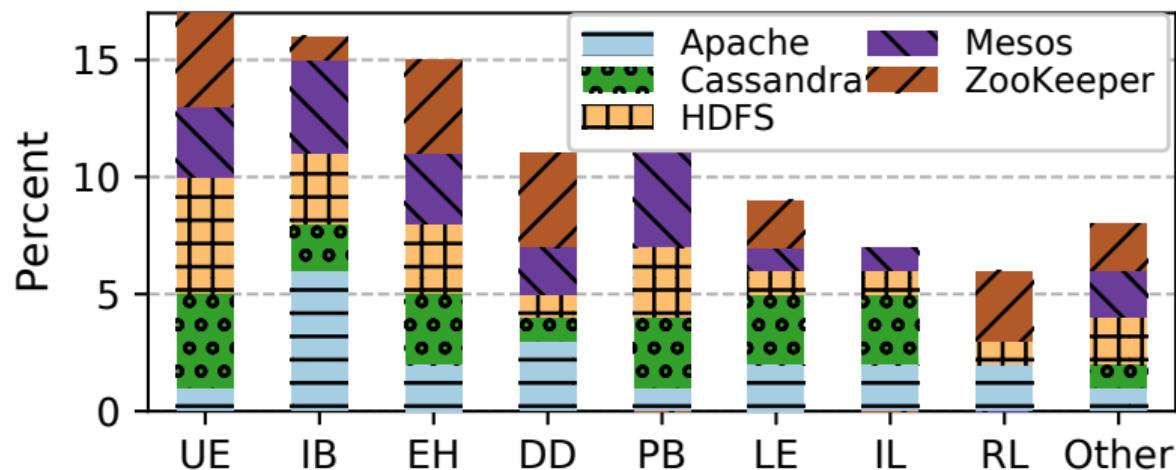
| Software  | Language | Cases | Versions           | Date Range            |
|-----------|----------|-------|--------------------|-----------------------|
| ZooKeeper | Java     | 20    | 17 (3.2.1–3.5.3)   | 12/01/2009–08/28/2018 |
| Cassandra | Java     | 20    | 19 (0.7.4–3.0.13)  | 04/22/2011–08/31/2017 |
| HDFS      | Java     | 20    | 14 (0.20.1–3.1.0)  | 10/29/2009–08/06/2018 |
| Apache    | C        | 20    | 16 (2.0.40–2.4.29) | 08/02/2002–03/20/2018 |
| Mesos     | C++      | 20    | 11 (0.11.0–1.7.0)  | 04/08/2013–12/28/2018 |

# Finding 1: Root Causes are Diverse

Root cause distribution

No single uniformed or dominating root cause<sup>2</sup>

Top three (total 48%) root cause types are uncaught errors, indefinite blocking, and buggy error handling

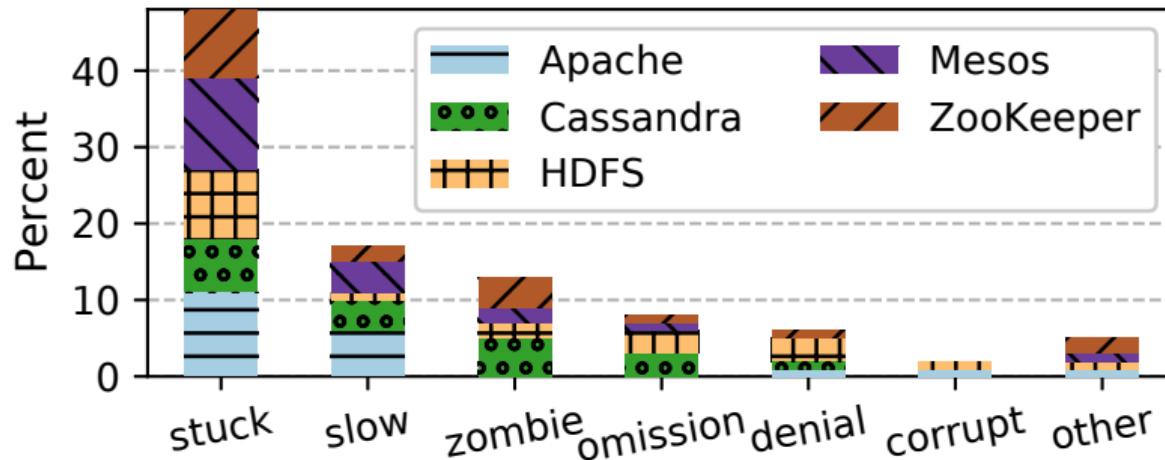


<sup>2</sup>UE: uncaught error; IB: indefinite blocking; EH: buggy error handling; DD: deadlock; PB: performance bug; LE: logic error; IL: infinite loop; RL: resource leak.

## Finding 2: Nearly Half Cases Cause **Stuck** Issues

Consequence

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



**17%** of the partial failures cause certain operations to take a long time to complete.  
(i.e. **slow**)

## Other Findings: Partial Failures are Hard to Detect

**15% of the partial failures are silent**

Including data loss, corruption, inconsistency, and wrong results

**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.**

**Debugging time is long**

The median diagnosis time is 6 days and 5 hours

**The majority (68%) of the failures are “sticky”**

The process will not recover from the faults by itself. The faulty process needs to be restarted or repaired to function again.

# Motivation

So how to detect and localize a partial failure in a big software?

What if we simply apply static or dynamic analysis?

## Static Analysis?

- ▶ no unique production env/workload
- ▶ unable to detect run-time problem

## Dynamic Analysis?

- ▶ existing detectors are too shallow
- ▶ unable to localize failures

Ask developers to manually add defensive checks?

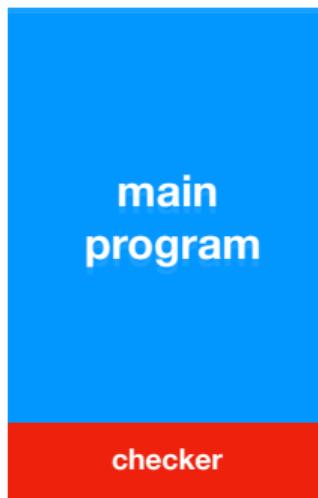
## Manual vs generated checkers

**Systematically** generated 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

# Intersection Principle

Construct customized checkers that **intersect** with the execution of a monitored process:



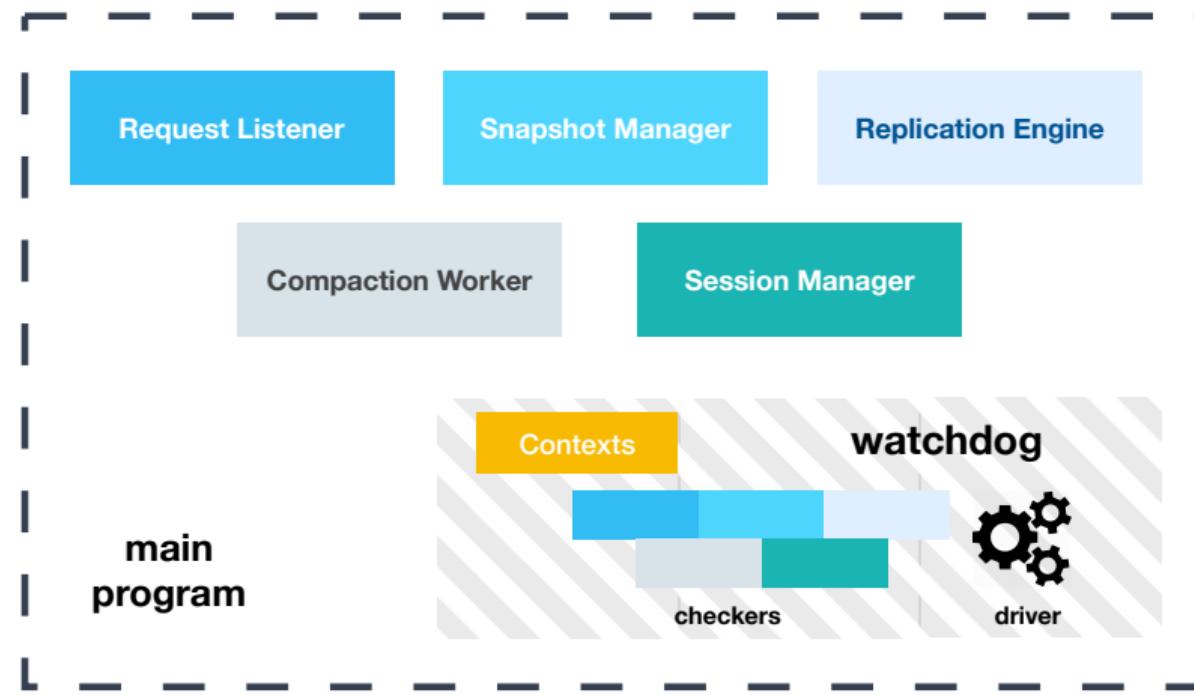
**existing approach**



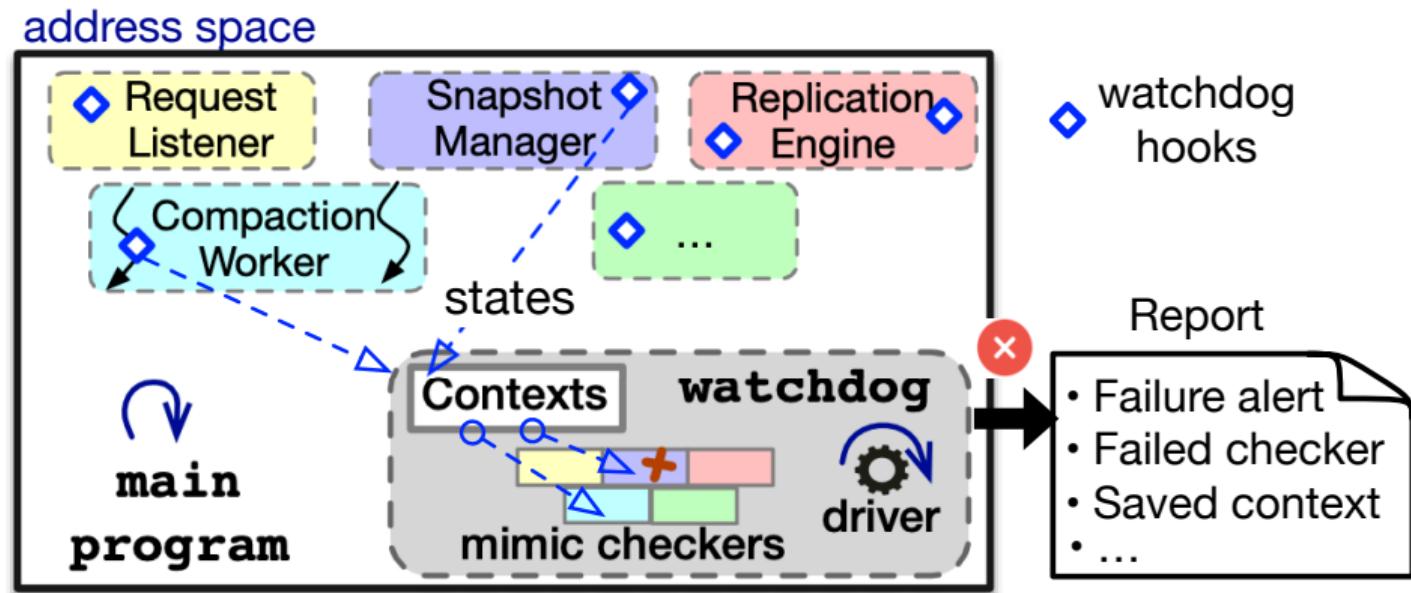
**our approach**

# Intrinsic watchdog: Runtime

An intrinsic watchdog is a dedicated monitoring extension for a process

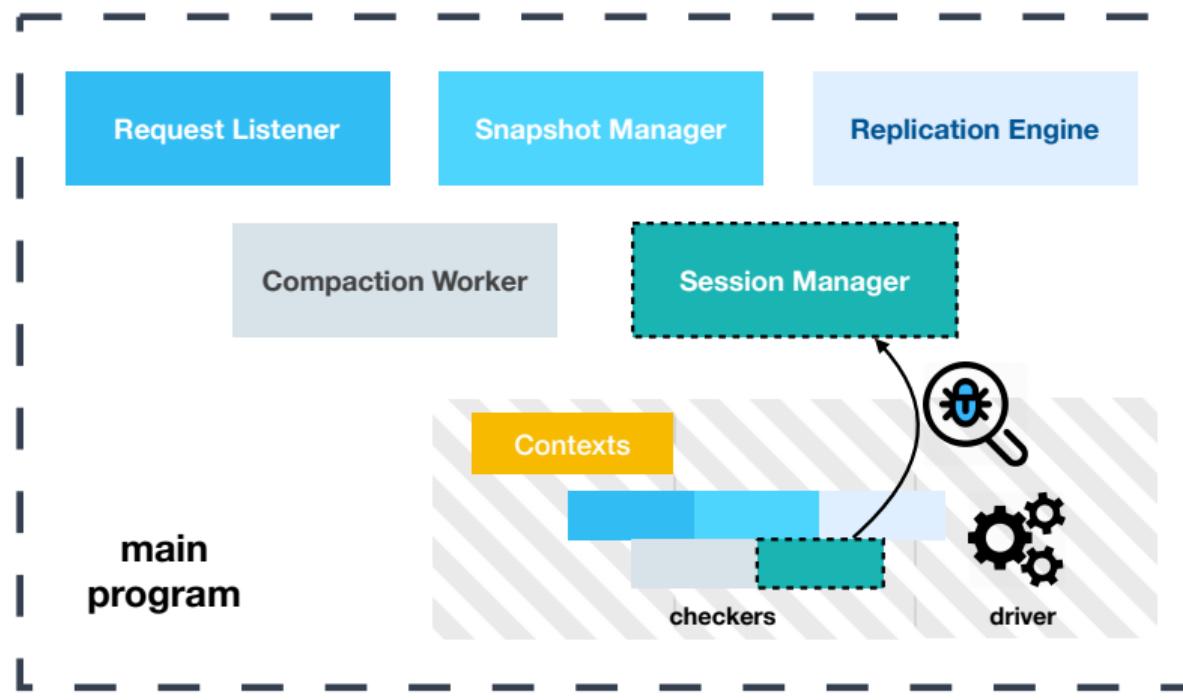


# Intrinsic watchdog: How it works?



## Characteristic I: Customized

- ▶ Regularly executes a set of checkers tailored to different modules
- ▶ Selects some representative operations from each module

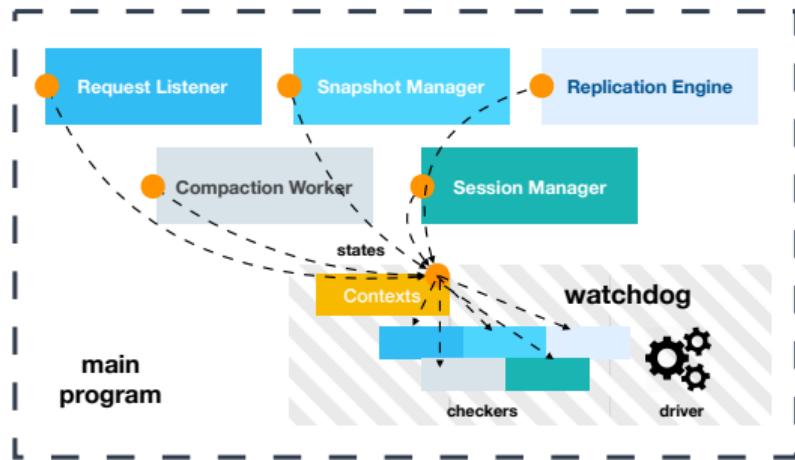


# Characteristic II: Stateful

To synchronized states, introduce

## Context

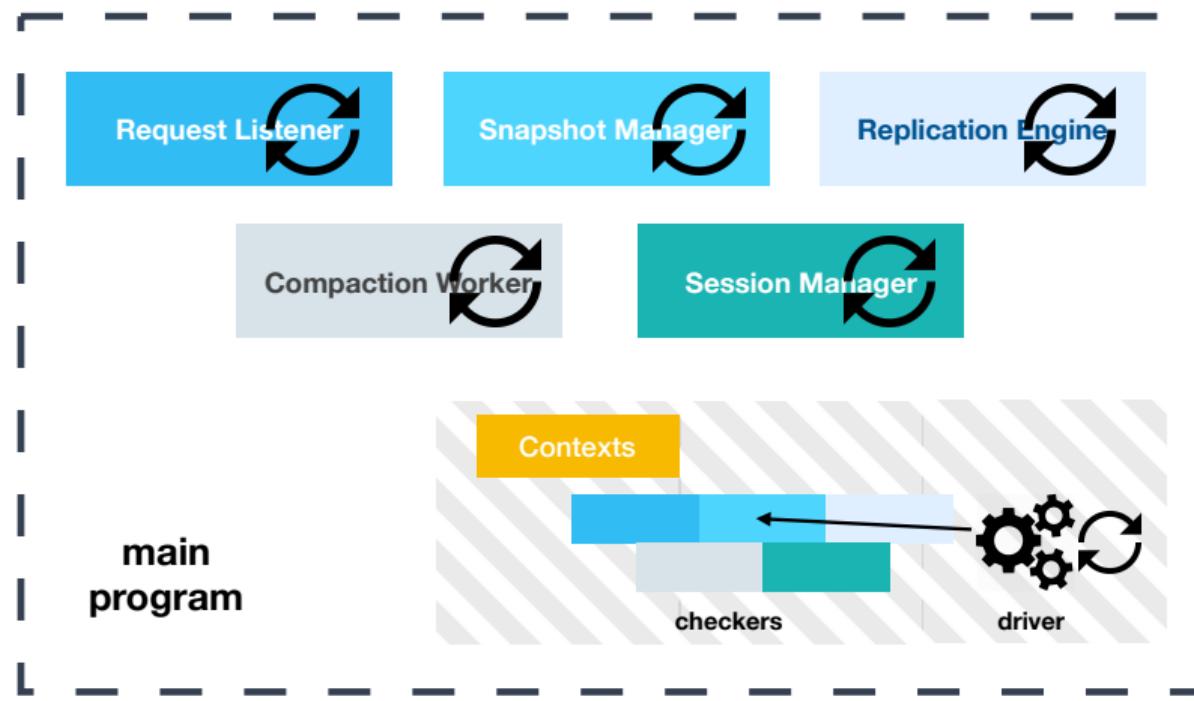
- ▶ bound to each checker
- ▶ holds all the arguments needed for the checker execution
- ▶ synchronized with the program state through hooks in the main program
- ▶ update with current state when hooks reached



**Note:** The watchdog driver will not execute a checker unless its context is ready.

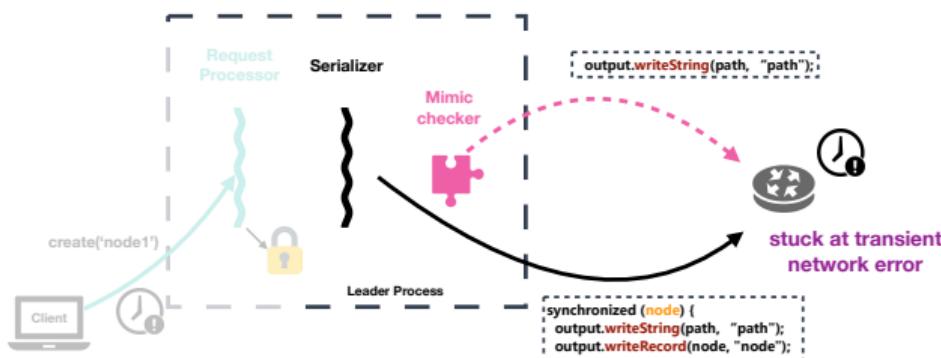
### Characteristic III: Concurrent

Run watchdog **concurrently** with the main program instead of **in-place** checking with **inserted** checkers



# Core Idea: Mimic Checking

Imitates some representative operations



**Exmaple:** Perform a similar operation (snapshot) and also get stuck at the same location

## Accuracy

- ▶ exercises code logic similar to the main program
- ▶ share execution environment in runtime
- ▶ increases coverage of checking targets
- ▶ can pinpoint the faulty module and failing instruction

# Implementation: OmegaGen

## Tool Overview

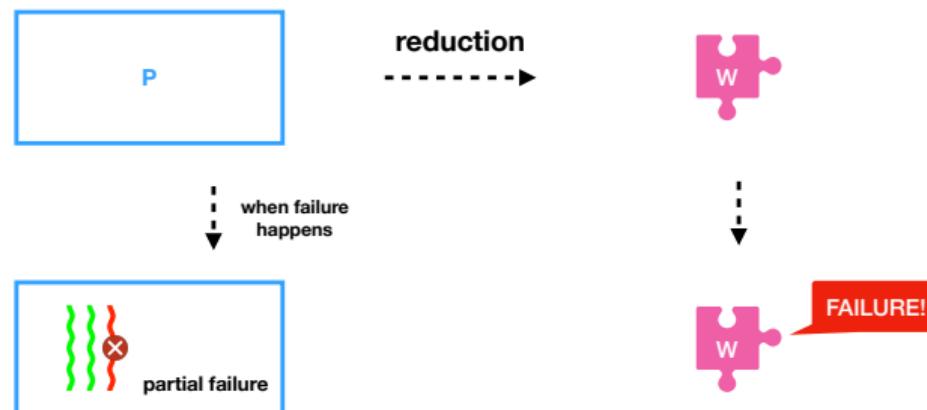
- ▶ a prototype that systematically generates mimic-type watchdogs for system softwares
- ▶ in Java with 8,100 SLOC, using Soot analysis framework
- ▶ **core technique:** program reduction

```
1 $ ./omegagen -jar zookeeper-3.4.6.jar -m zookeeper.manifest
2 analyzing..
3 generating..
4 instrumenting..
5 repackaging..
6 done. Total 1min 6s.
7 $ ls output/
8 zookeeper-3.4.6-with-wd.jar
```

# What is Program Reduction?

## Definition

Given a program  $P$ , create a watchdog  $W$  that can detect partial failures in  $P$  without imposing on  $P$ 's execution.



**Reduce:** We need not put everything into checkers, because a lot of operations are logically **deterministic** and should be checked before production. Only some of them are more **vulnerable** in the production environment.

# Program Reduction

For the source code of a given program, the process will go through 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

Identifies potentially long-running loops in the function body

e.g. `while(true), while(flag)`

However, an identified **long-running** loop may turn out to be **short-lived** in an actual run.

### **predicate**-based algorithm

a runtime property associated with a method which tracks whether a call site of this method is in fact reached

- ▶ insert a hook before the loop → sets its predicate
- ▶ insert a hook after the loop → unset its predicate
- ▶ pass caller's predicate set to callees

Runtime:

- ▶ activate or activates or deactivates the associated watchdog based on assigned predicate

# Step 1: Locate Long-running Regions

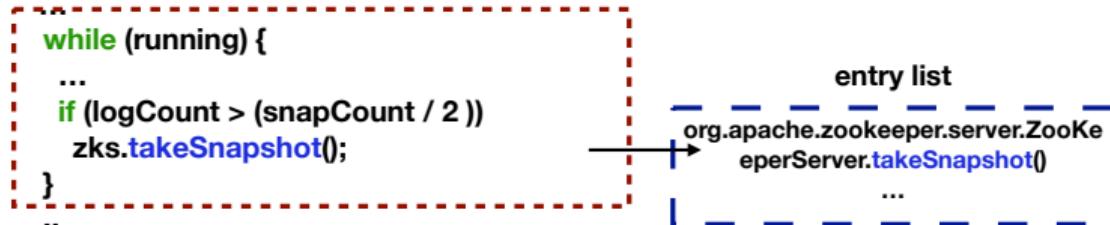
An example

initialization  
stage

long-running  
stage

cleanup  
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!");  
    }  
}
```



entry list

org.apache.zookeeper.server.ZooKeeperServer.takeSnapshot()

## Step 2: Reduce the Program

Recursively analyze each function to find out **vulnerable** operations (in the next step)

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

The code is annotated with red boxes and arrows:

- A red box surrounds the line `dt.serialize(oa, "tree");`. To its right, the text "keep reducing" is written.
- A red box surrounds the line `serializeNode(oa, new StringBuilder(""));`. To its right, the text "keep reducing" is written.
- A red arrow points from the word "keep reducing" after the first box down to the word "keep reducing" after the second box.

## Step 3: Locate Vulnerable Operations

Looks for potentially **vulnerable** operations in the control flow of those long-running methods.

- ▶ Heuristics (default):
  - ▶ synchronization
  - ▶ resource allocation
  - ▶ event polling
  - ▶ async waiting
  - ▶ invocations using external arguments
  - ▶ file or network I/
  - ▶ complex while loop conditional
- ▶ Customize rule table in configuration
- ▶ Developers can also explicitly annotate an operation as @vulnerable in source codes

## Step 4: Encapsulate Watchdog Checkers

Construct reduced method for each **vulnerable** method in main program

```
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");  
        ...  
    }  
}
```

**extracted vulnerable operations**

## Step 5: Insert Watchdog Hooks

To capture the real state of the main program in runtime and pass it to 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> childNodes = node.getChildren();
        if (childNodes != null)
            children = childNodes.toArray(new String[childNodes.size()]);
    }
    path.append('/');
    int off = path.length();
    ...
}
```

+ ContextFactory.serializeNode\_context\_setter(oa, node);

insert context hook before  
vulnerable operation

# An overview example

```
1 public class SyncRequestProcessor {
2     public void run() {
3         while (running) { ❶ identify long-running region
4             if (logCount > (snapCount / 2))
5                 zks.takeSnapshot(); ❷
6             ...
7         }
8     }
9 }
10 public class DataTree { ❸ reduce
11     public void serializeNode(OutputArchive oa, ...) {
12     ...
13     String children[] = null;
14     synchronized (node) { ❹ locate vulnerable operations
15         scount++;
16         oa.writeRecord(node, "node");
17         children = node.getChildren();
18     }
19     ...
20 } + ContextManger.serializeNode_reduced
21     _args_setter(oa, node);
22 } ❺ insert context hooks
```

(a) A module in main program

```
1 public class SyncRequestProcessor$Checker {
2     public static void serializeNode_reduced(
3         OutputArchive arg0, DataNode arg1) {
4         arg0.writeRecord(arg1, "node");
5     }
6     public static void serializeNode_invoke() {
7         Context ctx = ContextManger.❻ generate
8             serializeNode_reduced_context(); context
9         if (ctx.status == READY) {
10             OutputArchive arg0 = ctx.args_getter(0);
11             DataNode arg1 = ctx.args_getter(1);
12             serializeNode_reduced(arg0, arg1);
13         }
14     }
15     public static void takeSnapshot_reduced() {
16         serializeList_invoke();
17         serializeNode_invoke();
18     }
19     public static Status checkTargetFunction0() {
20         ... ❼ add fault signal checks
21         takeSnapshot_reduced();
22     }
23 }
```

(b) Generated checker

# Validate Impact of Caught Faults

## Transient or tolerable

- ▶ e.g. transient network delay that caused no damage

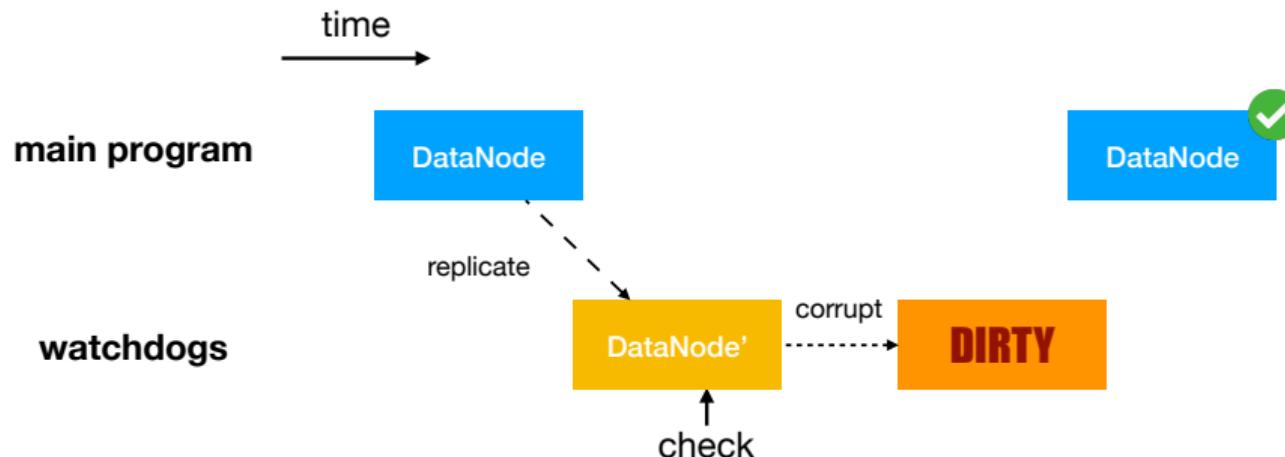
## Default: simply re-executes the checker and compare for transient errors

- ▶ allows developers write their own user-defined validation tasks to check some entry functions, e.g., `processRequest(req)`
- ▶ automate the part of deciding which validation task to invoke depending on which checker failed

# Prevent Side Effects

Context Replication (memory isolation)

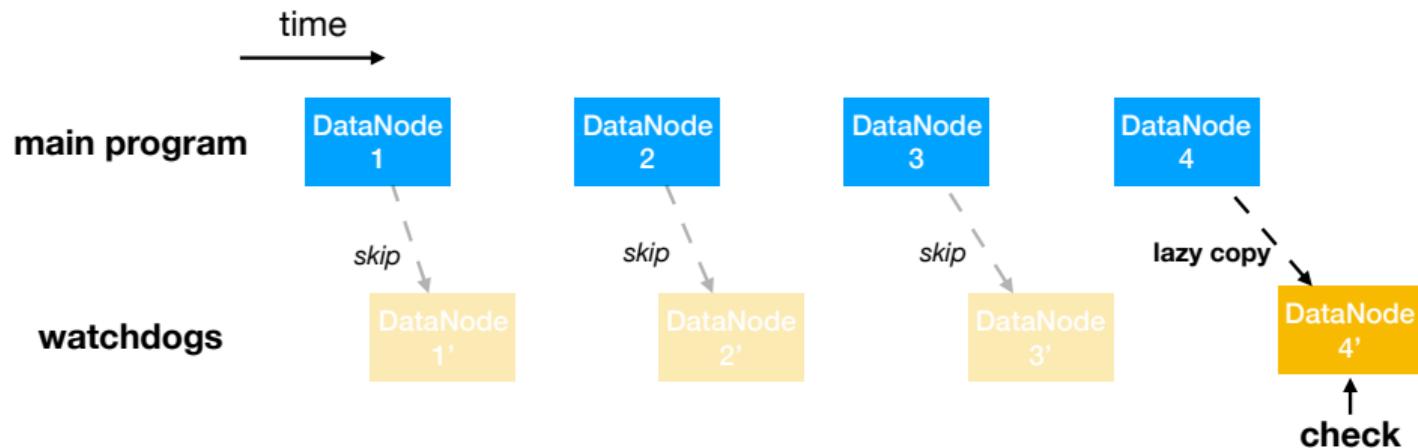
Context manager will replicate the checker context so that any modifications are contained in the watchdog's state



# Prevent Side Effects

Context Replication (memory isolation)

To reduce performance overhead: immutability analysis + lazy copy



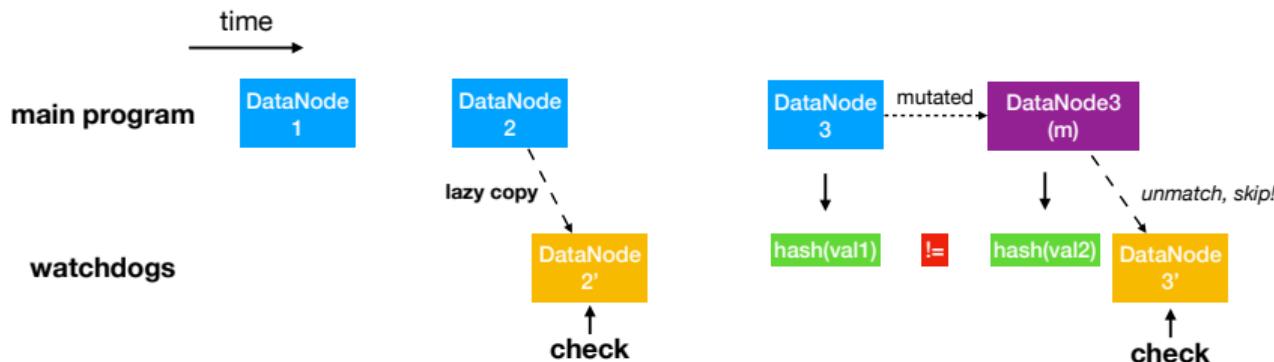
## Prevent Side Effects

## Context Replication (memory isolation)

Check consistency before copying and invocation with hashCode and versioning

Context attributes: version, weak\_ref and hash

- The lazy setter only sets these attributes without replicate the context.
  - If getter invoked, check `weak_ref != null`
  - Check if hash code of the referent's value matches `hash`

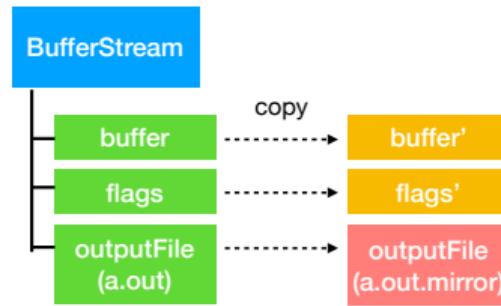


# Prevent Side Effects

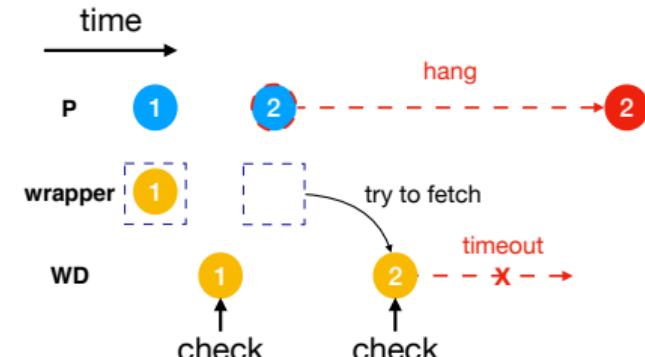
## I/O Redirection and Idempotent Wrappers (I/O isolation)

**write:** file-related resource replicated with target path changed to test file

**read:** let watchdogs pre-read contexts and cache



**write-redirection**



**read-redirection**

# Experiments

Scale & Generated watchdog

|            | 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 |
| 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    |

## 22 Real-world Partial Failures Reproduced for Evaluation I

| JIRA Id.       | Id. | Root Cause          | Conseq.   | Sticky? | Study? |
|----------------|-----|---------------------|-----------|---------|--------|
| ZooKeeper-2201 | ZK1 | Bad Synch.          | Stuck     | No      | Yes    |
| ZooKeeper-602  | ZK2 | Uncaught Error      | Zombie    | Yes     | Yes    |
| ZooKeeper-2325 | ZK3 | Logic Error         | Inconsist | Yes     | No     |
| ZooKeeper-3131 | ZK4 | Resource Leak       | Slow      | Yes     | Yes    |
| Cassandra-6364 | CS1 | Uncaught Error      | Zombie    | Yes     | Yes    |
| Cassandra-6415 | CS2 | Indefinite Blocking | Stuck     | No      | Yes    |
| Cassandra-9549 | CS3 | Resource Leak       | Slow      | Yes     | No     |
| Cassandra-9486 | CS4 | Performance Bug     | Slow      | Yes     | No     |
| HDFS-8429      | HF1 | Uncaught Error      | Stuck     | Yes     | Yes    |
| HDFS-11377     | HF2 | Indefinite Blocking | Stuck     | No      | Yes    |
| HDFS-11352     | HF3 | Deadlock            | Stuck     | Yes     | No     |

## 22 Real-world Partial Failures Reproduced for Evaluation II

| HDFS-4233      | HF4 | Uncaught Error        | Data Loss | Yes | No |
|----------------|-----|-----------------------|-----------|-----|----|
| HBase-18137    | HB1 | Infinite Loop         | Stuck     | Yes | No |
| HBase-16429    | HB2 | Deadlock              | Stuck     | Yes | No |
| HBase-21464    | HB3 | Logic Error           | Stuck     | Yes | No |
| HBase-21357    | HB4 | Uncaught Error        | Denial    | Yes | No |
| HBase-16081    | HB5 | Indefinite Blocking   | Silent    | Yes | No |
| MapReduce-6351 | MR1 | Deadlock              | Stuck     | Yes | No |
| MapReduce-6190 | MR2 | Infinite Loop         | Stuck     | Yes | No |
| MapReduce-6957 | MR3 | Improper Err Handling | Stuck     | Yes | No |
| MapReduce-3634 | MR4 | Uncaught Error        | Zombie    | Yes | No |
| Yarn-4254      | YN1 | Improper Err Handling | Stuck     | Yes | No |

# Baseline

**Client Panorama**<sup>3</sup>: instrument and monitor client responses

**Probe Falcon**<sup>4</sup>: daemon thread in the process that periodically invokes internal functions with synthetic requests

**Signal** script that scans logs and checks JMX metrics

**Resource** daemon thread that monitors memory usage, disk and I/O health, and active thread count

---

<sup>3</sup>Peng Huang et al. "Capturing and enhancing in situ system observability for failure detection". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 1–16.

<sup>4</sup>Joshua B Leners et al. "Detecting failures in distributed systems with the falcon spy network". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 279–294.

# Detection Time

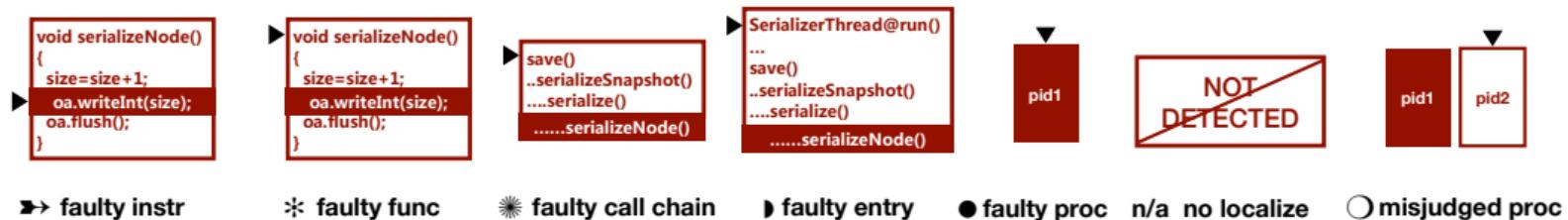
- median detection time: 4.2s
- 12 by default vulnerable operation rules
- 8 by system-specific rules
- effective for liveness issues like deadlock, indefinite blocking, and safe issues
- less effective for silent correctness errors

|        | ZK1  | ZK2   | ZK3  | ZK4   | CS1   | CS2  | CS3    | CS4   | HF1   | HF2  | HF3  | HF4  | HB1   | HB2  | HB3  | HB4  | HB5  | MR1  | MR2  | MR3  | MR4  | YN1  |
|--------|------|-------|------|-------|-------|------|--------|-------|-------|------|------|------|-------|------|------|------|------|------|------|------|------|------|
| Client | x    | 2.47  | 2.27 | x     | 441   | x    | x      | x     | x     | x    | x    | x    | x     | 4.81 | x    | 6.62 | x    | x    | x    | x    | 8.54 | 7.38 |
| Probe  | x    | x     | x    | x     | 15.84 | x    | x      | x     | x     | x    | x    | x    | x     | 4.71 | x    | 7.76 | x    | x    | x    | x    | x    | x    |
| Signal | 12.2 | 0.63  | 1.59 | 0.4   | 5.31  | x    | x      | x     | x     | x    | x    | 0.77 | 0.619 | x    | 0.62 | 61.0 | x    | x    | x    | x    | 0.60 | 1.16 |
| Res.   | 5.33 | 0.56  | 0.72 | 17.17 | 209.5 | x    | -19.65 | x     | -3.13 | x    | x    | 0.83 | x     | x    | x    | 0.60 | x    | x    | x    | x    | x    | x    |
| 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 | x    | 0.80 | 5.89 | 1.01 | 4.07 | 1.46 | 4.68 | x    |

# Detection Localization

|        | ZK1 | ZK2 | ZK3 | ZK4 | CS1 | CS2 | CS3 | CS4 | HF1 | HF2 | HF3 | HF4 | HB1 | HB2 | HB3 | HB4 | HB5 | MR1 | MR2 | MR3 | MR4 | YN1 |     |
|--------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Client | n/a | •   | •   | n/a | •   | n/a | •   | n/a | ○   | n/a | n/a | n/a | n/a | •   | •   |     |
| Probe  | n/a | n/a | n/a | n/a | ▶   | n/a | ▶   | n/a | ▶   | n/a | n/a | n/a | n/a | n/a | n/a |     |
| Signal | •   | ▶   | •   | •   | ▶   | n/a | n/a | n/a | n/a | ▶   | ▶   | n/a | ●   | ●   | n/a | n/a | n/a | n/a | ▶   | ▶   |     |     |     |
| Res.   | •   | •   | •   | •   | •   | n/a | •   | n/a | •   | n/a | n/a | •   | n/a | •   | n/a | n/a | •   | n/a | n/a | n/a | n/a | n/a |     |
| Watch  | ▶   | ▶   | •   | *   | ▶   | *   | •   | *   | *   | ●   | ▶   | ▶   | ▶   | ▶   | ▶   | n/a | ▶   | *   | ▶   | ▶   | *   | ▶   | n/a |

Failure localization for the real-world cases.



# False Alarms

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

- validator → the watchdog false alarm ratios decrease

|          | ZooKeeper | Cassandra | HDFS       | HBase    | MapReduce   | Yarn     |
|----------|-----------|-----------|------------|----------|-------------|----------|
| 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%       |
| 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%       |

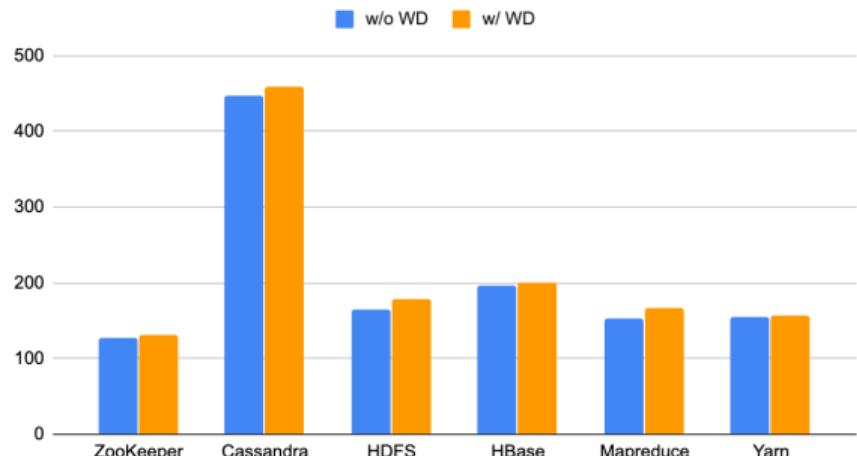
# System Overhead

Time to generate watchdogs: 17 min for HBase, < 5 min for the others

Throughput (op/s) w/ different checkers

|                 | ZooKeeper | Cassandra | HDFS | HBase | MapReduce | Yarn |
|-----------------|-----------|-----------|------|-------|-----------|------|
| Base            | 428.0     | 3174.9    | 90.6 | 387.1 | 45.0      | 45.0 |
| w/ Probe.       | 417.6     | 3128.2    | 89.4 | 374.3 | 44.9      | 44.9 |
| w/<br>Resource. | 424.8     | 3145.4    | 89.9 | 385.6 | 44.9      | 44.6 |
| w/ Watch.       | 399.8     | 3014.7    | 85.1 | 366.4 | 42.1      | 42.3 |

Heap memory usage w/ and w/o watchdogs



5.0%-6.6% throughput overhead w/  
watchdog

4.3% (avg) memory overhead w/ watchdog

# By-product: Discovering A New Partial Failure Bug<sup>5</sup>

Watchdogs report the failure in 4.7 seconds occasionally during experiments

```
[...] Start to check for watchdog[ 66/96 ]
[...] Ready to run checker:
org.apache.zookeeper.server.DataTree@void
serializeAcls(org.apache.jute.OutputArchive)
[...] Start to check for interfaceinvoke
a_.<org.apache.jute.OutputArchive: void
writeInt(int,java.lang.String)>($r0, "map")
[...] Try to clone for writeInt $1, writeInt $2
[...] Status: TIMEOUT. Description: Execution time
exceeds threshold.
[...] Context Index: 11763
[...] Start to validate captured error..
```

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

<sup>5</sup>Detailed report available at <https://issues.apache.org/jira/browse/ZOOKEEPER-3531>

# Summary

- ▶ **Motivation:** Modern software are increasingly complex and often fail partially
  - ▶ these partial failures cannot be detected by process-level failure detectors
- ▶ **Case study:** On 100 partial failure cases
  - ▶ No main root causes, hard to detect with existing methods
- ▶ **Proposed solution:** 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

# Thanks