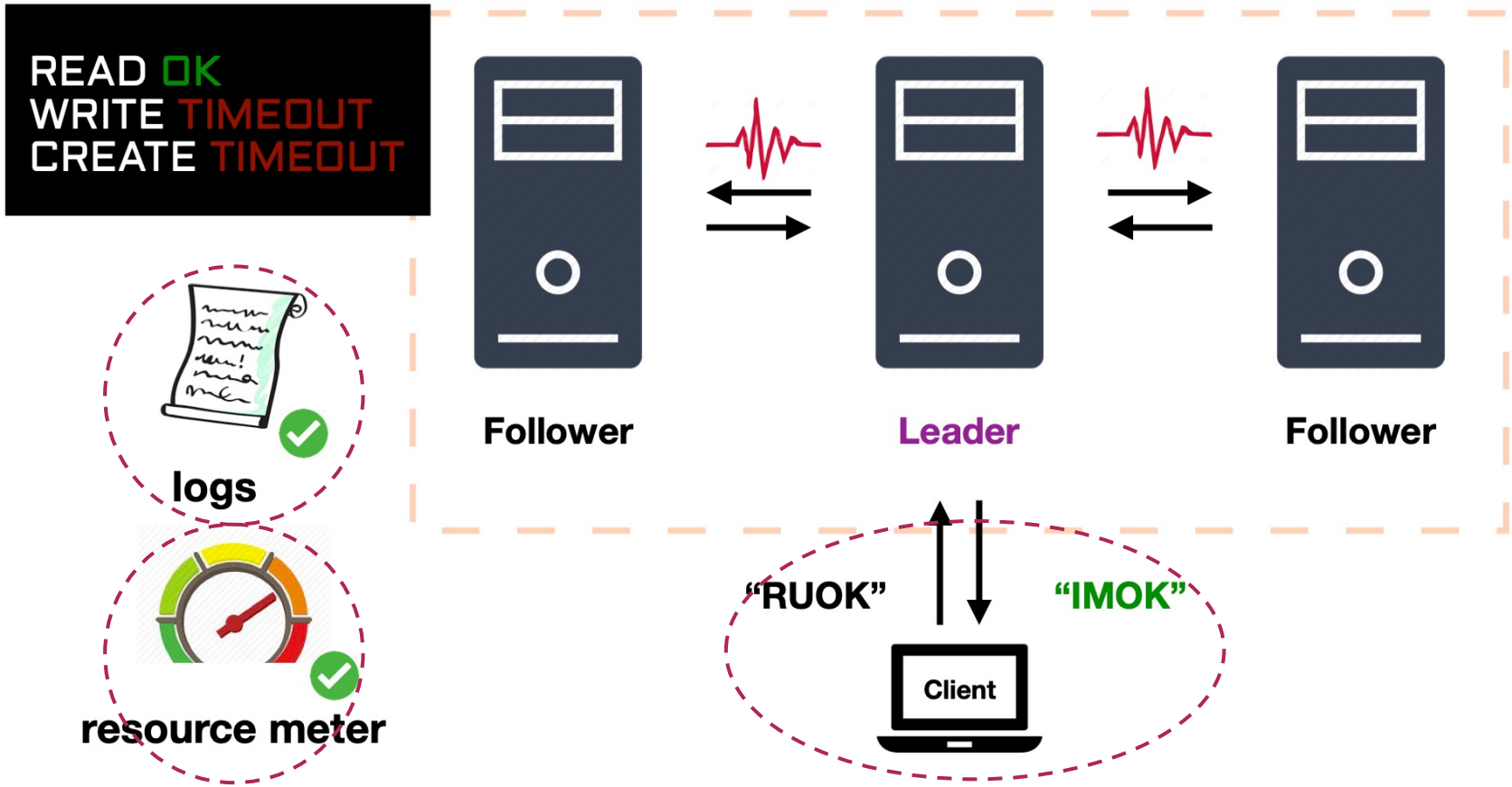
# Understanding, Detecting and Localizing Partial Failures in Large System Software

[NSDI’20] Chang Lou etc.

# Understanding, Detecting and Localizing Partial Failures in Large System Software

# Partial Failures

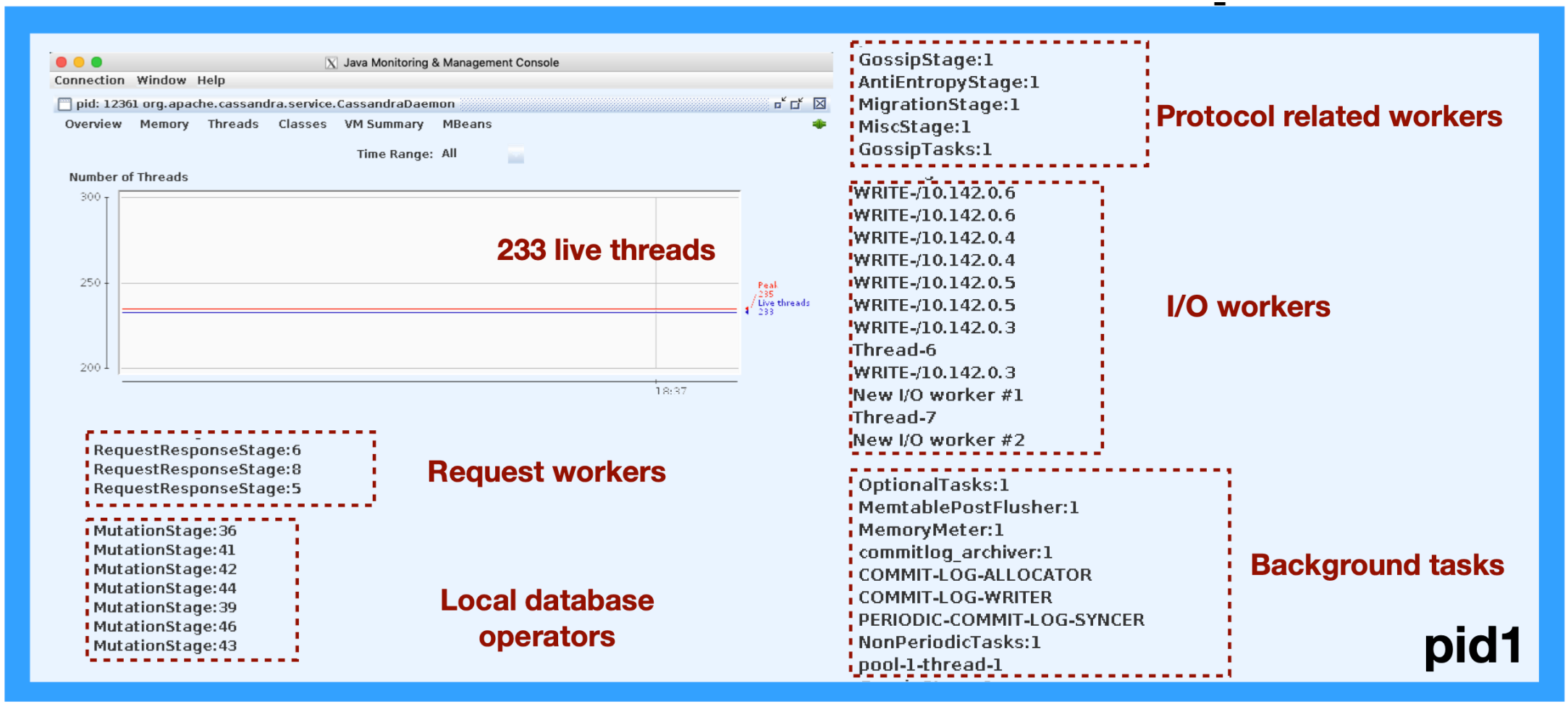
- A fault not crash system but causes safety or liveness violation or server slowness




Hard to detect

# Modern Software Suffers Partial Failrues

- Modern software is complex so that is easily subject to partial failures



# Understanding, Detecting and Localizing Partial Failures in Large System Software

# Study Methodology

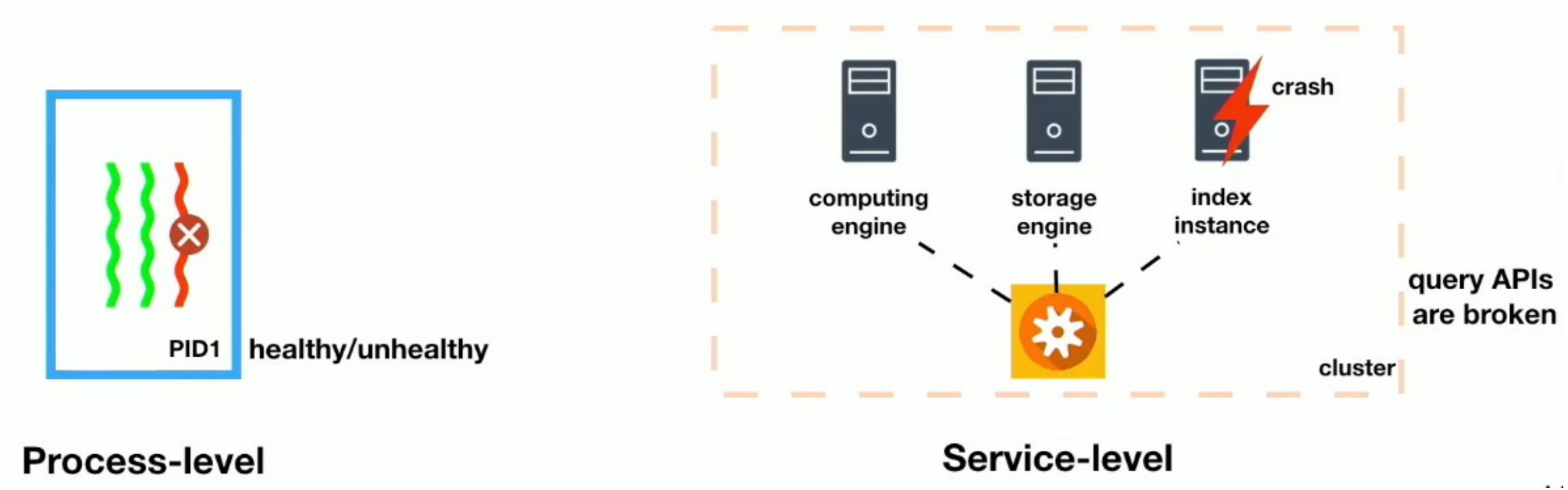
- 100 partial failure cases from five large, widely used software systems

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

| 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 |

# Study Scope

- Process-level



# Main Findings

- The root causes of studied failures are diverse (Figure 2)
  - Top three (total 48%) root cause types:  uncaught errors, indefinite blocking, and buggy error handling

- The Majority of failures violate liveness (Figure 3)
  - Functionality stuck (48%), slow operations (17%)

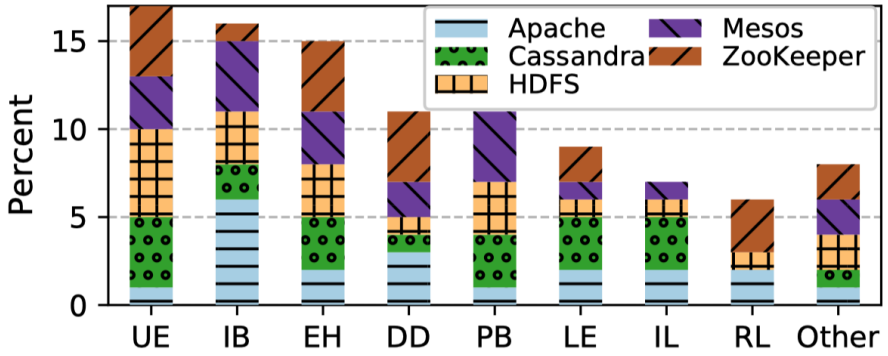



**Figure 2:** Root cause distribution. *UE*: uncaught error; *IB*: indefinite blocking; *EH*: buggy error handling; *DD*: deadlock; *PB*: performance bug; *LE*: logic error; *IL*: infinite loop; *RL*: resource leak.

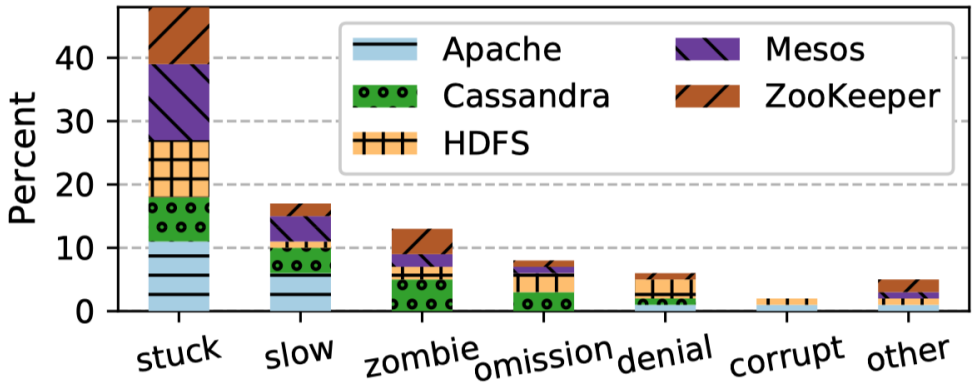
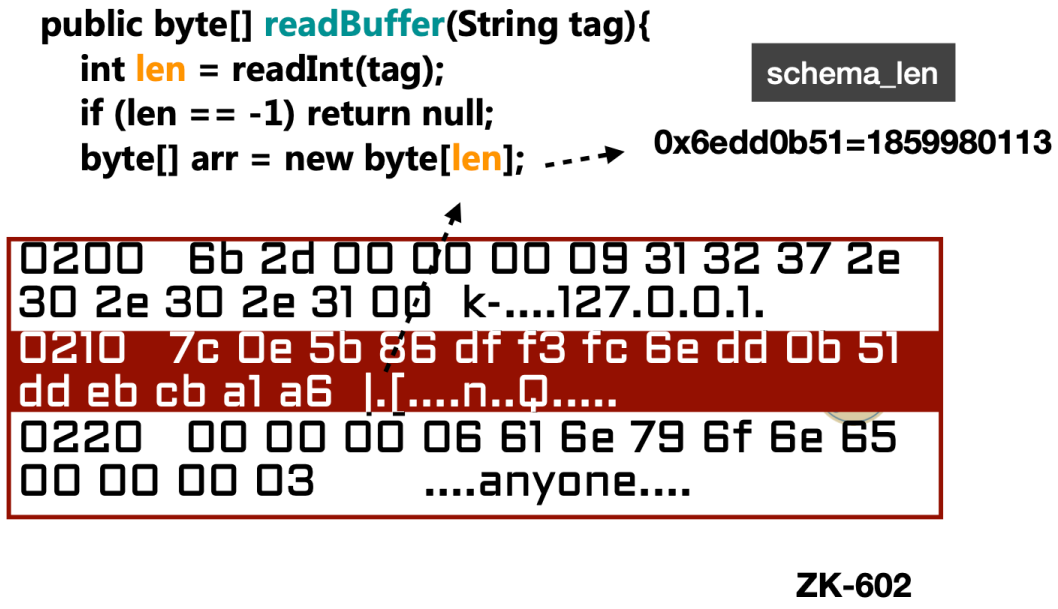



**Figure 3:** Consequence of studied failures.

# Main Findings

- Most cases (71%) triggered by unique production workload or environment




- Long debugging time
  - The median diagnosis time is 6 days and 5 hours

# Features of Partial Failures

- Common in modern software
- Long debugging time

→ Certainly need detecting and localizing

- Production-dependent

→ Static approaches could not work well

- Diverse root causes

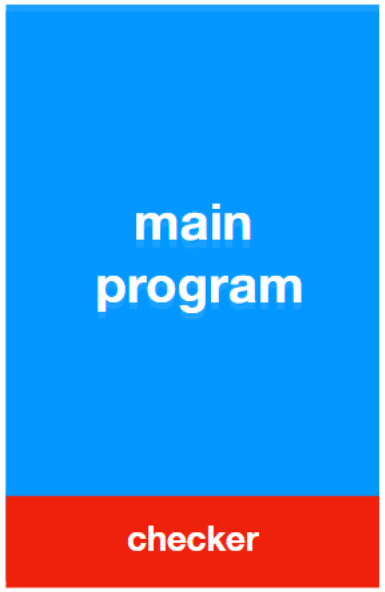
→ Manually writing checker is unrealistic

**Automatically construct runtime checkers**

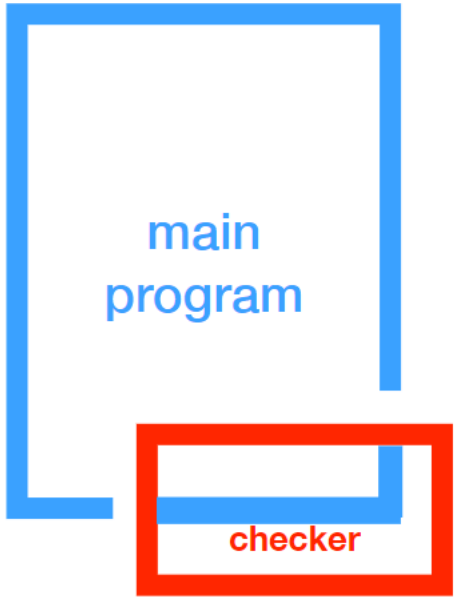
# Understanding, Detecting and Localizing Partial Failures in Large System Software

# Design Principle: Intersecting with Main Program

- Partial failures typically involve specific software feature and bad states




existing approach



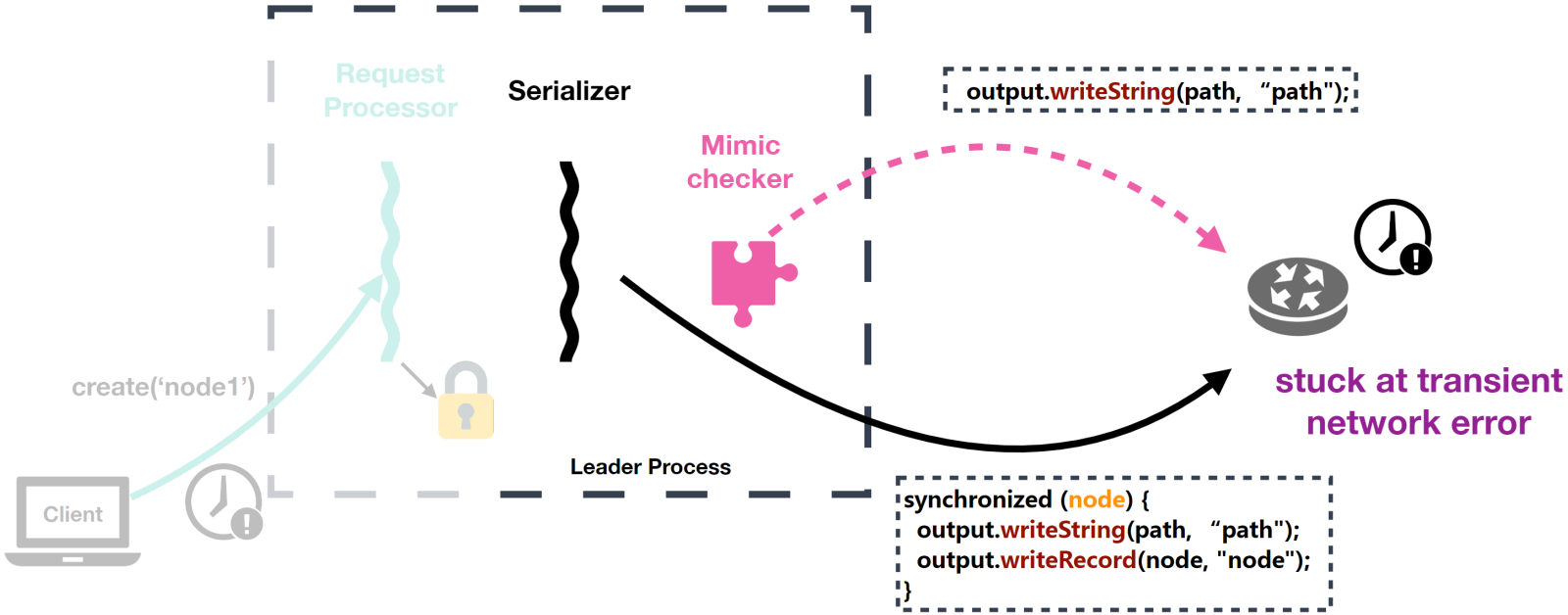

our approach

# Three Characteristics of Runtime Checker

- Customized: tailored to monitored modules
- Stateful: entailing specific input and program state
- Concurrent: decoupled with main program without delaying its executions

# Core Idea: Mimic Checker

- Imitating representative operations in main program and detecting errors

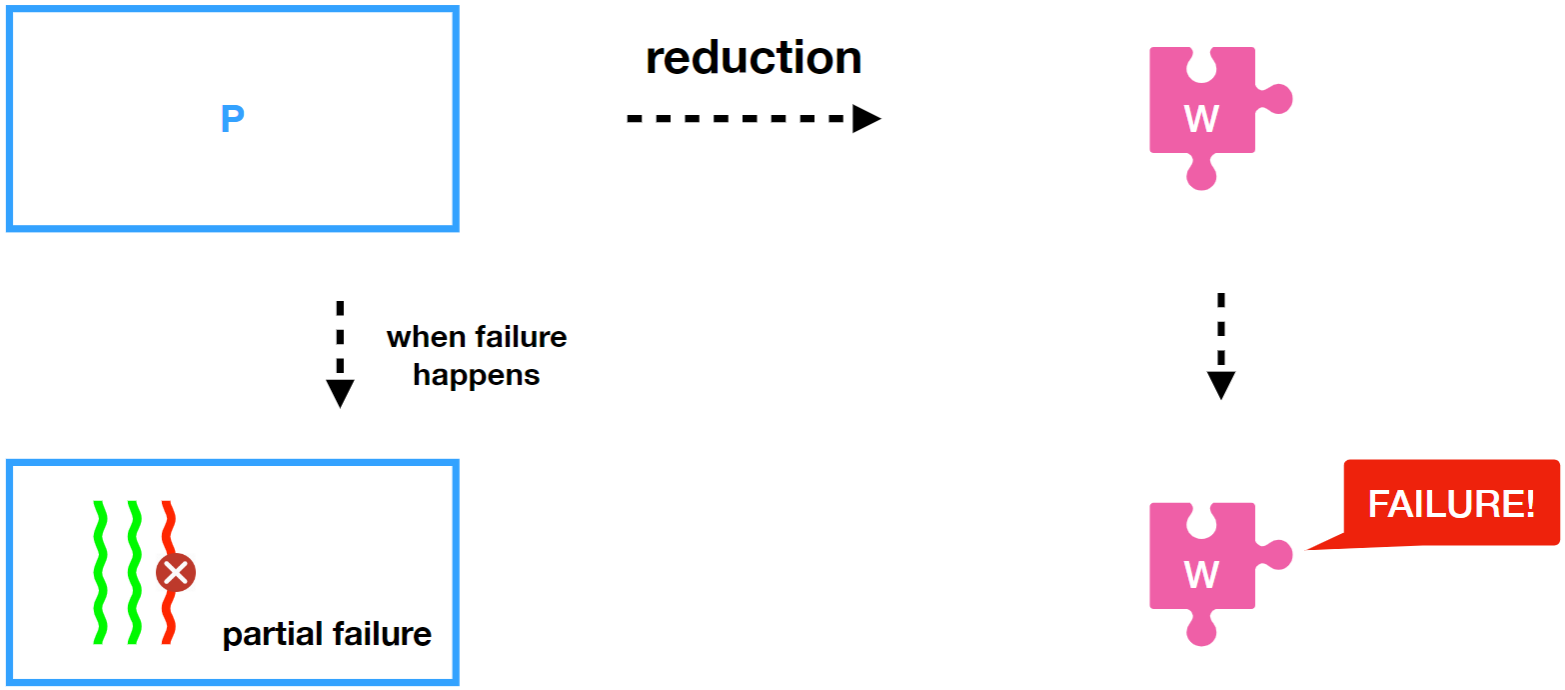



Reflect the monitored process' status

Pinpoint the faulty module and failing instruction

# Obtain Mimic Checker by Program Reduction

- Selecting vulnerable operations and encapsulating them into checkers

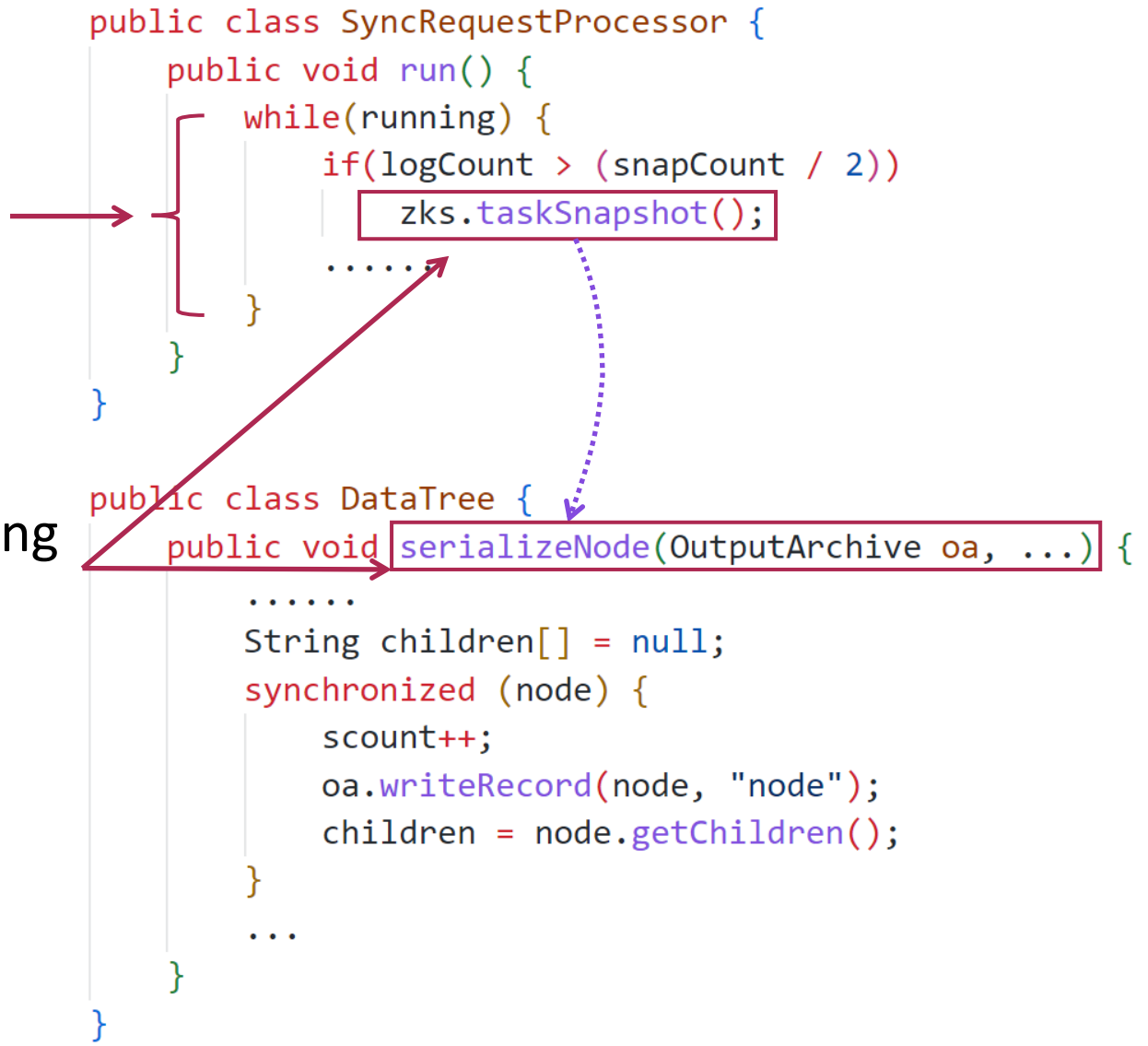


# Seven Steps for generating checkers

1. Identify Long-running Methods
2. Locate Vulnerable Operations
3. Reduce Main Program
4. Encapsulate Reduced Program
5. Add Checks to Catch Faults
6. Validate Impact of Caught Faults
7. Prevent Side Effects

# 1.Identify Long-running Methods

- Identifying long-running loops in each call graph node
- Coloring invocation in long-running loops recursively



# 2.Locate Vulnerable Operations

- Use heuristics to locate vulnerable operations in long-running methods

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

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

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

long-running method

# 3.Reduce Main Program

- Top-down reduction: retaining vulnerable operations

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

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

```
public class SyncRequestProcessor$Checker {
    public static void takeSnapshot_reduced() {
        serializeNode_reduced();
    }

    public static void serializeNode_reduced(
        OutputArchive arg0, DataNode arg1){
        arg0.writeRecord(arg1, "node");
    }
}
```

# 4.Encapsulate Reduced Program

- Analyzing and initializing arguments before reduced methods

```
public class SyncRequestProcessor$Checker {
    public static void takeSnapshot_reduced() {
        serializeNode_reduced();
    }

    public static void serializeNode_reduced(
        OutputArchive arg0, DataNode arg1){
        arg0.writeRecord(arg1, "node");
    }
}
```

Not executable due to missing definitions

```
public class SyncRequestProcessor$Checker {
    public static void takeSnapshot_reduced() {
        serializeNode_invoke();
    }

    public static void serializeNode_invoke(){
        Context ctx = ContextManager.serializeNode_reduced_context();
        if(ctx.status == READY){
            OutputArchive arg0 = ctx.args_getter(0);
            DataNode arg1 = ctx.args.getter(1);
            serializeNode_reduced(arg0, arg1);
        }
    }

    public static void serializeNode_reduced(
        OutputArchive arg0, DataNode arg1){
        arg0.writeRecord(arg1, "node");
    }
}
```

Get values of arguments from context

# 4.Encapsulate Reduced Program

- Instrumenting main program to store contexts

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

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

```
public class SyncRequestProcessor$Checker {
    public static void takeSnapshot_reduced() {
        serializeNode_invoke();
    }

    public static void serializeNode_invoke(){
        Context ctx = ContextManager.serializeNode_reduced_context();
        if(ctx.status == READY){
            OutputArchive arg0 = ctx.args_getter(0);
            DataNode arg1 = ctx.args.getter(1);
            serializeNode_reduced(arg0, arg1);
        }
    }

    public static void serializeNode_reduced(
        OutputArchive arg0, DataNode arg1){
        arg0.writeRecord(arg1, "node");
    }
}
```

# 5.Add Checks to Catch Faults

- Safety checking relies on explicit error signals (assertions, exceptions, and error code)
- Liveness checking relies on timers
- $wd\ \_assert$ API for semantics

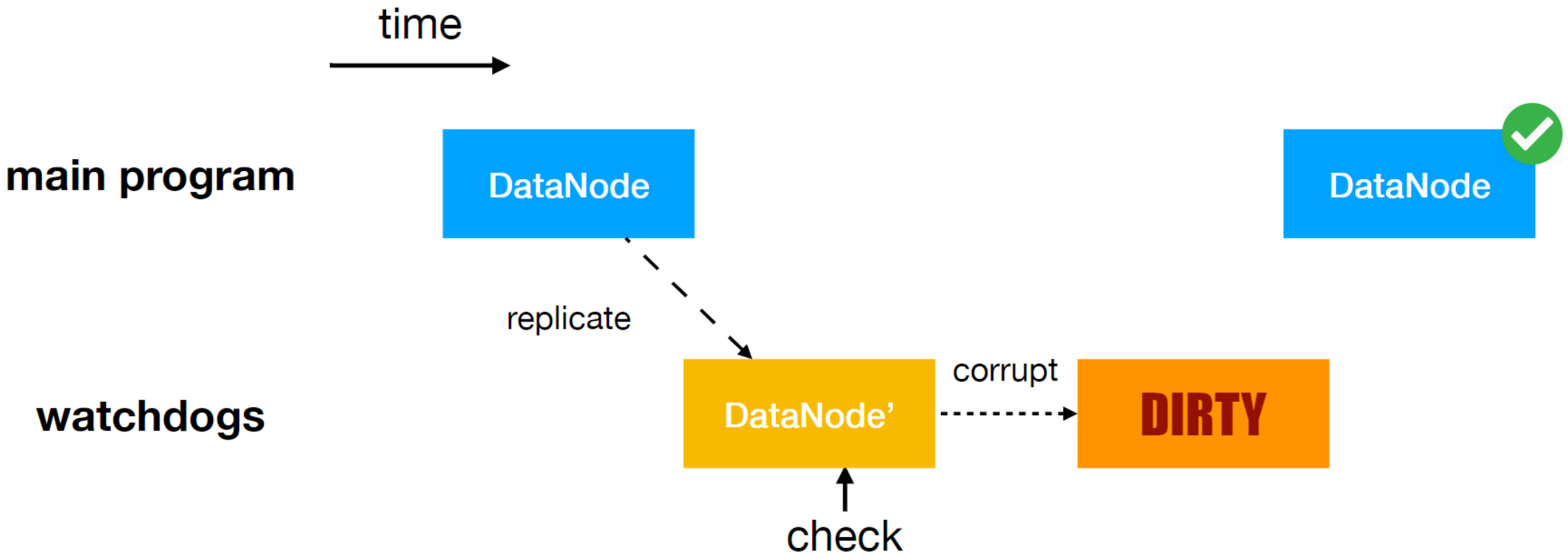
```
public class SyncRequestProcessor$Checker {
    public static void takeSnapshot_reduced() {
        serializeNode_invoke();
    }

    public static void serializeNode_invoke(){
        Context ctx = ContextManager.serializeNode_reduced_context();
        if(ctx.status == READY){
            OutputArchive arg0 = ctx.args_getter(0);
            DataNode arg1 = ctx.args.getter(1);
            serializeNode_reduced(arg0, arg1);
        }
    }

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

    public static Status checkTargetFunction0(){
        long startTime = System.currentTimeMillis();
        try{
            takeSnapshot_reduced();
        } catch(Exception e){
            safetyViolation();
        }
        long timeCost = System.currentTimeMillis() - startTime;
        if(timeCost > threshold){
            livenessViolation();
        }
    }
}
```

# 6.Validate Impact of Caught Faults

- A reported error may be transient or tolerant
  - e.g. a transient network delay that caused no damage
- Simply re-executing the checker and compare for transient errors
- Providing validation task skeletons :
  - Allowing developers write their own user-defined validation tasks
  - Automatically deciding which validation task to invoke depending on which checker failed.

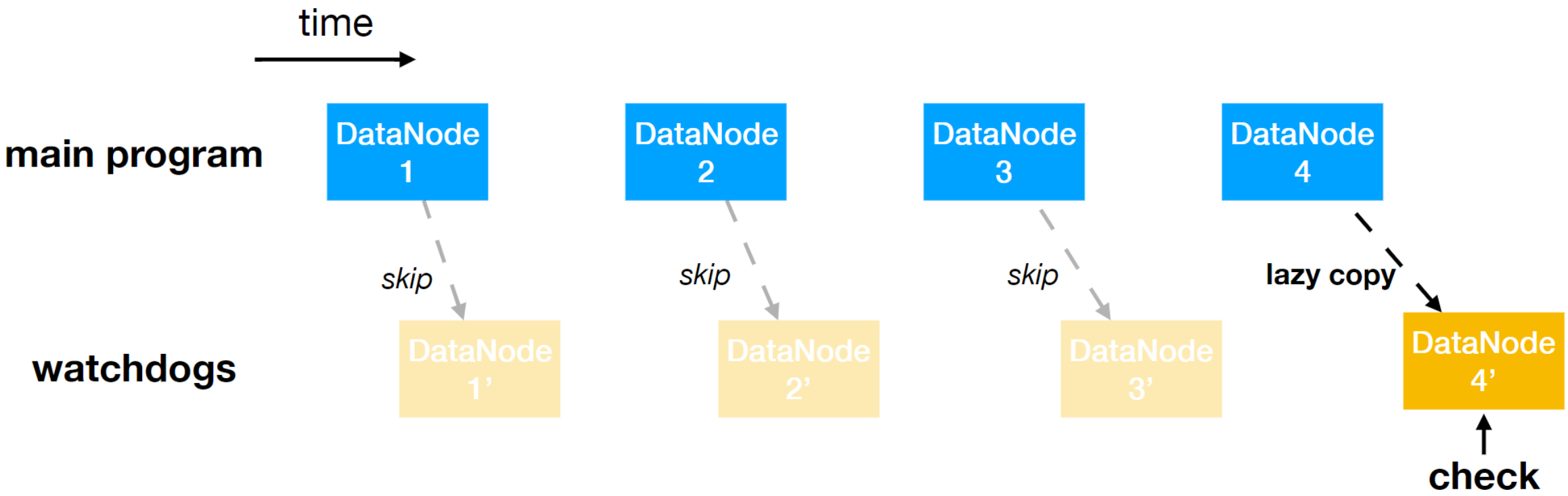
# 7.Prevent Side Effects

- Context Replication
  - Preventing checkers from modifying the main program's states



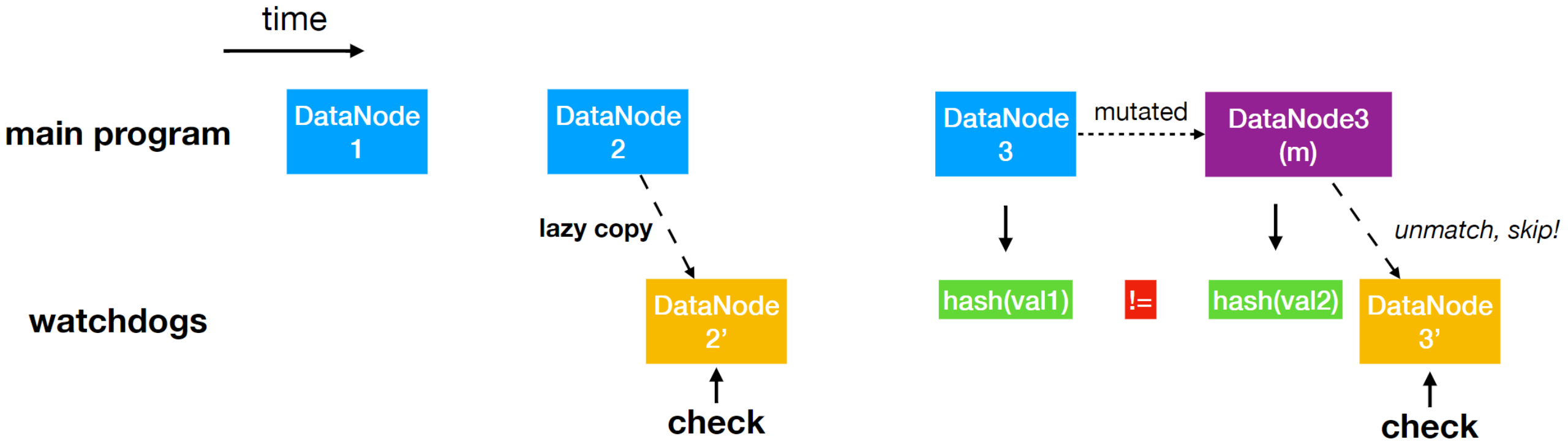
# 7.Prevent Side Effects

- Context Replication
  - To reduce performance overhead: immutability analysis + lazy copy



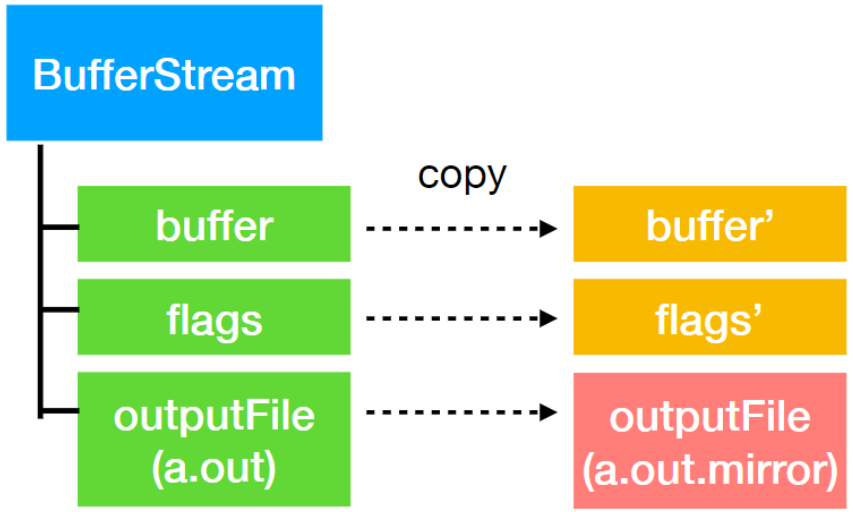
# 7.Prevent Side Effects

- Context Replication
  - Checking consistency before copying and invocation with hashCode and versioning

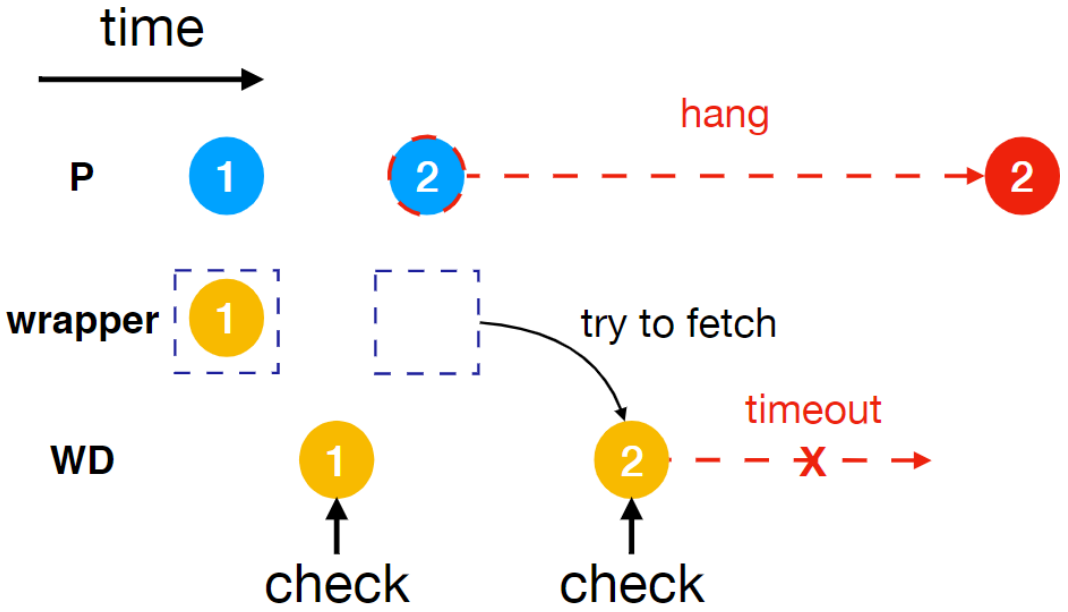


# 7.Prevent Side Effects

- I/O Redirection and Idempotent Wrappers (I/O isolation)
  - Write: file-related resource replicated with target path changed to test file
  - Read: caching read results of main program




write-redirection




read-redirection

# Evaluation

# Evaluated Systems

- Evaluate OmegaGen on six large systems
  - ZK: ZooKeeper; CS: Cassandra; HF: HDFS; HB: HBase; MR: MapReduce; YN: Yarn

| | ZK | CS | HF | HB | MR | YN |
|---|---|---|---|---|---|---|
| SLOC | 28K | 102K | 219K | 728K | 191K | 229K |
| Methods | 3,562 | 12,919 | 79,584 | 179,821 | 16,633 | 10,432 |

# Generated Watchdogs

- Watchdog generated by **static** analysis

- 60% of threads in production contains at least one watchdog checker

| | ZK | CS | HF | HB | MR | YN |
|---|---|---|---|---|---|---|
| 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

| Id. | Root Cause | Conseq. | Sticky? | Study? |
|---|---|---|---|---|
| ZK1 [45] | Bad Synch. | Stuck | No | Yes |
| ZK2 [66] | Uncaught Error | Zombie | Yes | Yes |
| ZK3 [47] | Logic Error | Inconsist. | Yes | No |
| ZK4 [48] | Resource Leak | Slow | Yes | Yes |
| CS1 [7] | Uncaught Error | Zombie | Yes | Yes |
| CS2 [8] | Indefinite Blocking | Stuck | No | Yes |
| CS3 [12] | Resource Leak | Slow | Yes | No |
| CS4 [11] | Performance Bug | Slow | Yes | No |
| HF1 [29] | Uncaught Error | Stuck | Yes | Yes |
| HF2 [24] | Indefinite Blocking | Stuck | No | Yes |
| HF3 [23] | Deadlock | Stuck | Yes | No |
| HF4 [28] | Uncaught Error | Data Loss | Yes | No |
| HB1 [20] | Infinite Loop | Stuck | Yes | No |
| HB2 [19] | Deadlock | Stuck | Yes | No |
| HB3 [22] | Logic Error | Stuck | Yes | No |
| HB4 [21] | Uncaught Error | Denial | Yes | No |
| HB5 [18] | Indefinite Blocking | Silent | Yes | No |
| MR1 [35] | Deadlock | Stuck | Yes | No |
| MR2 [34] | Infinite Loop | Stuck | Yes | No |
| MR3 [36] | Improper Err Handling | Stuck | Yes | No |
| MR4 [33] | Uncaught Error | Zombie | Yes | No |
| YN1 [44] | Improper Err Handling | Stuck | Yes | No |

# Detection overview

- Baseline Detectors
  - Built-in detectors (heartbeat) in the six systems cannot handle partial failures as all
  - Implement four types of advanced detectors

| Detector | Description |
|---|---|
| Client(Panorama [OSDI ’18]) | instrument and monitor client responses |
| Probe(Falcon [SOSP ’11]) | 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 |

# Detecting Overview

- Methodology
  - Run checks every second
- Result
  - 20 out of 22 cases was detected

| | 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 | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ |

# Detection Localization

- Six level of decreasing accuracy
  - ➳ faulty instruction
  - ✻ faulty function
  - ✺ faulty call chain
  - ◗ faulty entry
  - ● faulty process
  - ❍ misjudged process


- 55%(11/20) of the detected cases are pinpointed to faulty instruction

| | ZK1 | ZK2 | ZK3 | ZK4 | CS1 | CS2 | CS3 | CS4 | HF1 | HF2 | HF3 | HF4 | HB1 | HB2 | HB3 | HB4 | HB5 | MR1 | MR2 | MR3 | MR4 | YN1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Watchdog | ➳ | ➳ | ● | ✻ | ➳ | ✻ | ● | ✻ | ✻ | ✺ | ➳ | ➳ | ➳ | ➳ | n/a | ➳ | ✺ | ➳ | ➳ | ✺ | ➳ | n/a |
| Client | n/a | ● | ● | n/a | ● | n/a | n/a | n/a | n/a | n/a | 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 | ◗ | 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 | n/a | n/a | ➳ | ➳ |
| Resource | ● | ● | ● | ● | ● | n/a | ● | n/a | ● | n/a | n/a | ● | n/a | n/a | n/a | ● | n/a | n/a | n/a | n/a | n/a | n/a |

# More Detection Results

- Random fault-injection tests
  - Experiment on ZooKeeper
  - Inject four types of faults
    - Infinite loop
    - Arbitrary delay
    - System resource contention
    - I/O delay
  - Trigger 16 synthetic failures and 13 of them was detected

- A new bug
  - In Zookeeper version v3.5.5
  - Different from bugs in reproduction failures cases

# False Alarms

- False alarms emerge when random nodes restart which leads to socket connection errors or resource contention

- Validator mechanism significantly reduce false alarm ratio

| | 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 |

# Limitations

- vulnerable operation analysis is heuristics-based
- generated watchdogs are ineffective to catch silent semantic failures
- Achieve memory isolation with static analysis-assisted context replication
- OmegaGen generates watchdogs to report failures for individual process
- watchdogs focus on fault detection and localization but not recovery

# Conclusion

- Modern software systems become ever more complex
  - Existing solutions cannot capture partial failures among them
- This work research partial failures via case studies
- OmegaGen : a static analysis tool that automatically generates customized checkers
  - Prove effective on reproduction databases
  - Report a new bug

# Thanks

Comments are welcome!