Understanding, Detecting and Localizing Partial Failures in Large System Software[1]

October 16, 2020

[1] 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

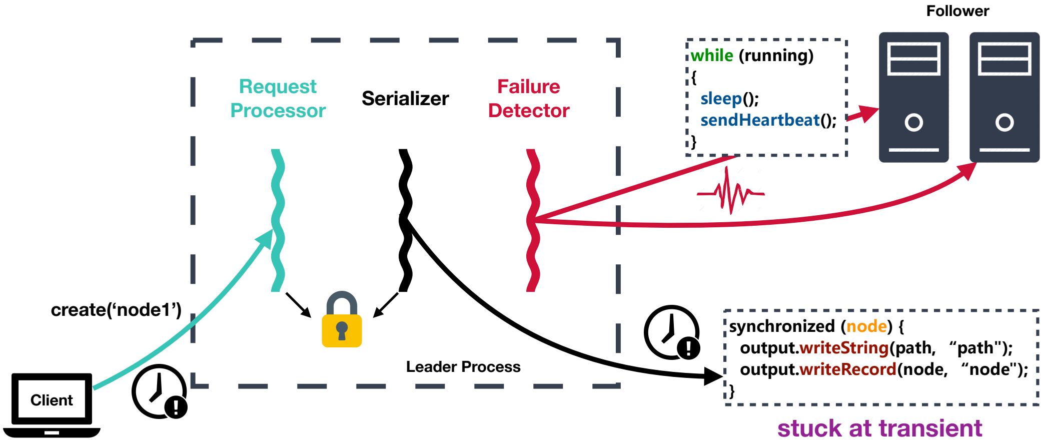
Motivation

Proposed Design
- Ideas
- Implementation

Evaluation

What is a Partial Failure?

An Example



```
while (running)
{
  sleep();
  sendHeartbeat();
}
```

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

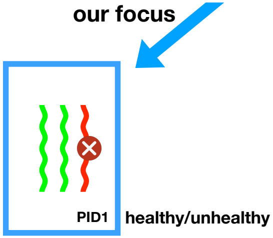
stuck at transient network error

What is a Partial Failure?

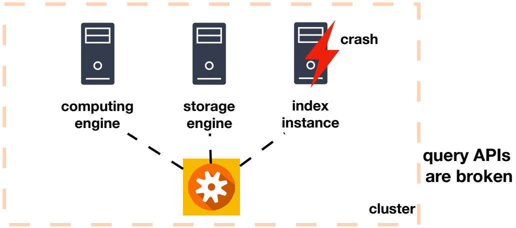
Definition

A partial failure is, in a process π to be when a fault **does not** crash π 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.



Process-level



Service-level

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)*

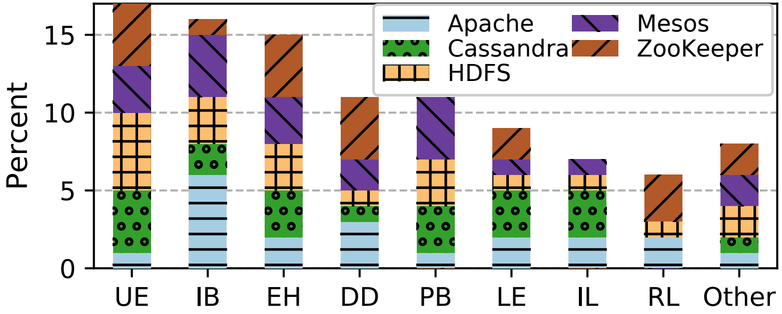
Software	Language	Cases	Versions	Date Range
ZooKeeper	Java	20	17 (3.2.1–3.5.3)	12/01/2009–08/28/2018
Cassandra	Java	20	19 (0.7.4–3.0.13)	04/22/2011–08/31/2017
HDFS	Java	20	14 (0.20.1–3.1.0)	10/29/2009–08/06/2018
Apache	C	20	16 (2.0.40–2.4.29)	08/02/2002–03/20/2018
Mesos	C++	20	11 (0.11.0–1.7.0)	04/08/2013–12/28/2018

Finding 1: Root Causes are Diverse

Root cause distribution

No single uniformed or dominating root cause[2]

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

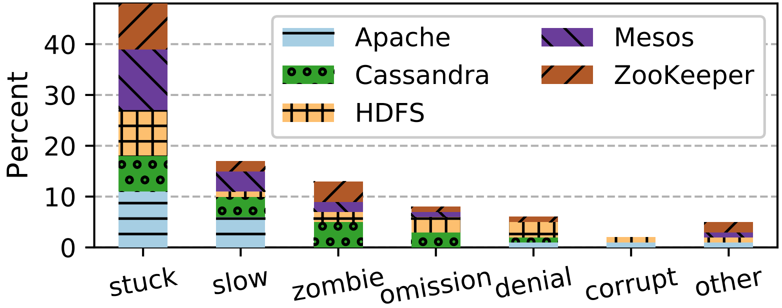


[2]UE: uncaught error; IB: indefinite blocking; EH: buggy error handling; DD: deadlock; PB: performance bug; LE: logic error; IL: infinite loop; RL: resource leak.

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?

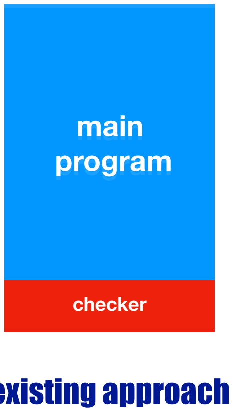
Manual vs generated checkers

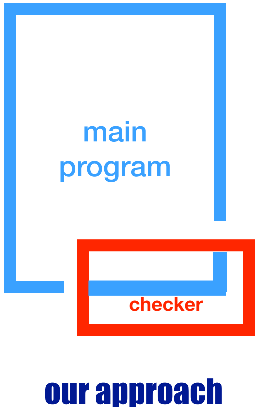
Systematically generated checkers to ease developers' burden

- 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

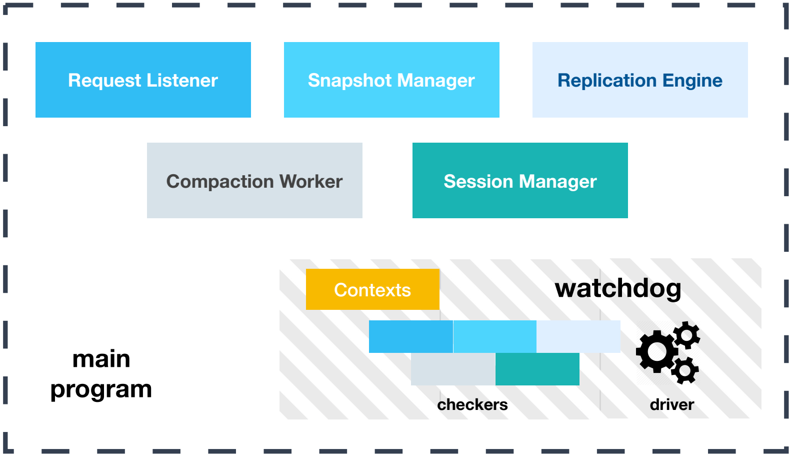
Construct customized checkers that **intersect** with the execution of a monitored process:



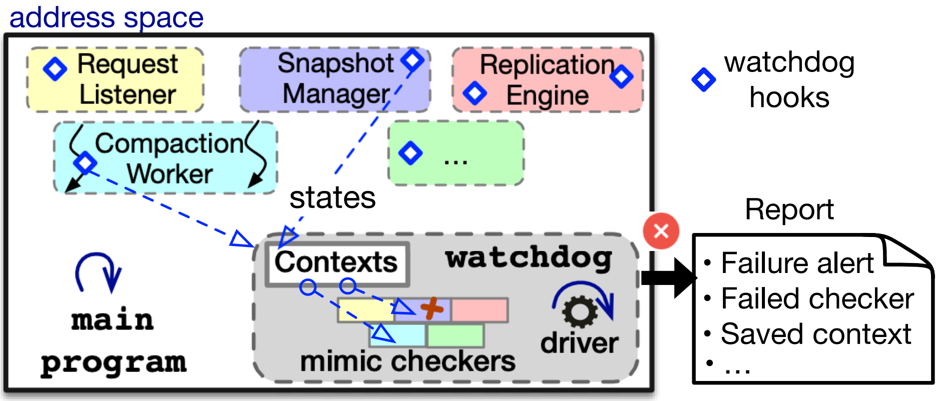


Intrinsic watchdog: Runtime

An intrinsic watchdog is a dedicated monitoring extension for a process

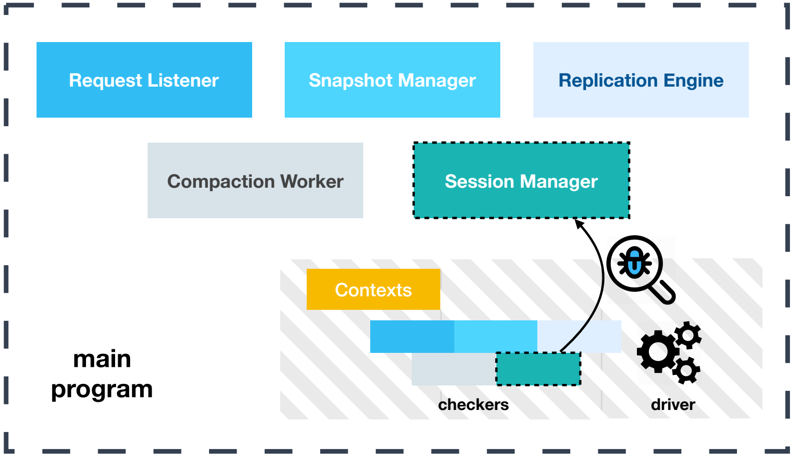


Intrinsic watchdog: How it works?



Characteristic I: Customized

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

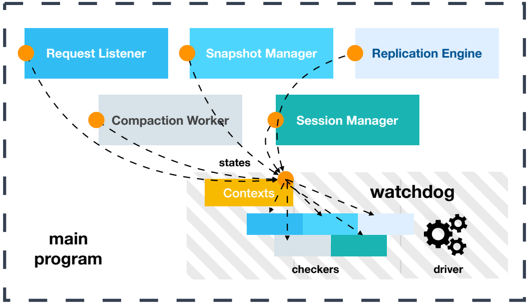


Characteristic II: Stateful

To synchronized states, introduce

Context

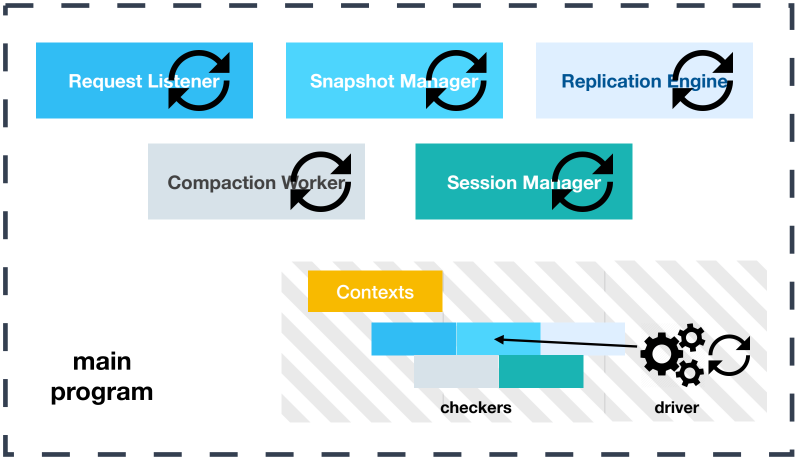
- bound to each checker
- holds all the arguments needed for the checker execution
- synchronized with the program state through hooks in the main program
- update with current state when hooks reached



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

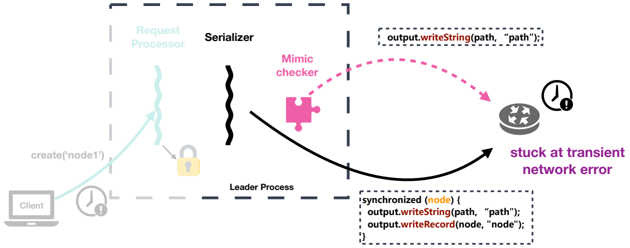
Characteristic III: Concurrent

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



Core Idea: Mimic Checking

Imitates some representative operations



```
output.writeString(path, "path");
```

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

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

Accurancy

- 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

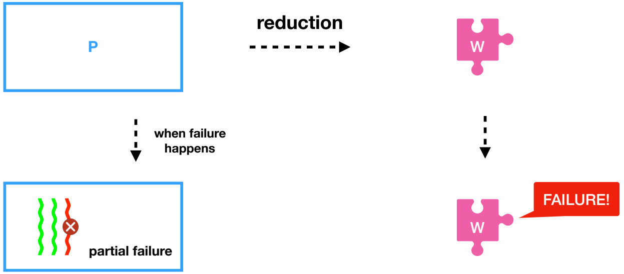
- a prototype that systematically generates mimic-type watchdogs for system softwares
- in Java with 8,100 SLOC, using Soot analysis framework
- **core technique:** program reduction

```
$ ./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?

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

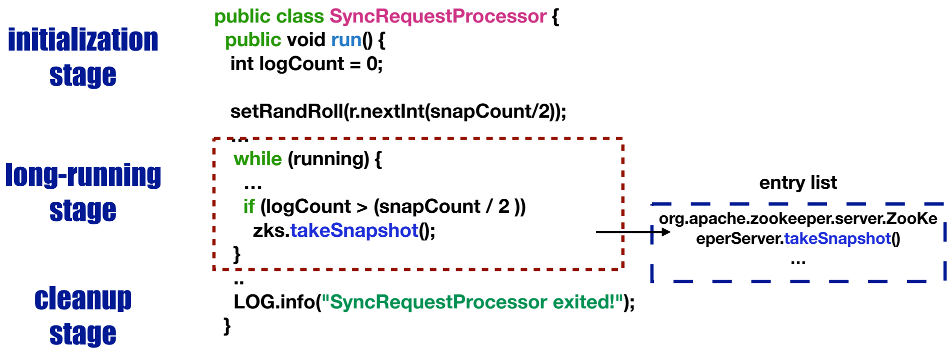
- insert a hook before the loop $\rightarrow$ sets its predicate
- insert a hook after the loop $\rightarrow$ unset its predicate
- pass caller's predicate set to callees

Runtime:

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

Step 1: Locate Long-running Regions

An example



```
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!");
 }
```

```
org.apache.zookeeper.server.ZooKe
eperServer.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”);                                keep reducing
  }
 }
public class DataTree{
 public void serialize(OutputArchive oa, String tag) {
   scout = 0;
   serializeNode(oa, new StringBuilder(“”));                keep reducing
   ...
 }
```

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

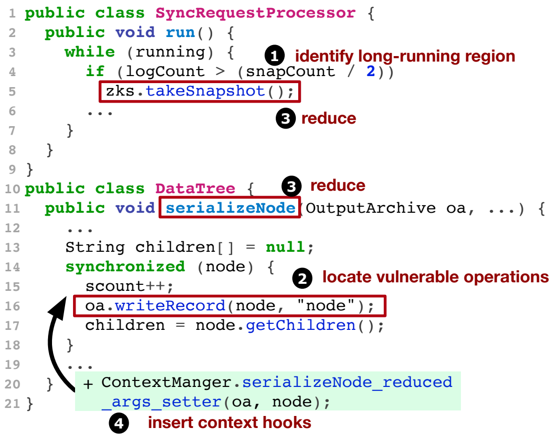
Step 5: Insert Watchdog Hooks

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);
                                        + ContextFactory.serializeNode_context_setter(oa, node);
    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();
    ...
}
```

insert context hook before vulnerable operation

An overview example

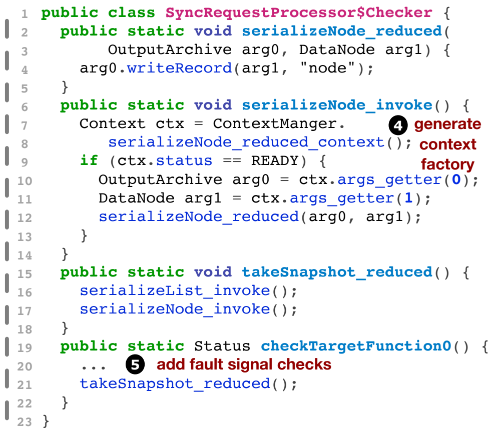


```
public class SyncRequestProcessor {
  public void run() {
    while (running) {
      if (logCount > (snapCount / 2))
        zks.takeSnapshot();
      ...
    }
  }
}
public class DataTree {
  public void serializeNode(OutputArchive oa, ...) {
    ...
    String children[] = null;
    synchronized (node) {
      scount++;
      oa.writeRecord(node, "node");
      children = node.getChildren();
    }
    ...
  }
}
```

```
+ ContextManger.serializeNode_reduced
  _args_setter(oa, node);
```

1 identify long-running region
3 reduce
3 reduce
2 locate vulnerable operations
4 insert context hooks

(a) A module in main program



```
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);
    }
  }
  public static void takeSnapshot_reduced() {
    serializeList_invoke();
    serializeNode_invoke();
  }
  public static Status checkTargetFunction0() {
    ...
    takeSnapshot_reduced();
  }
}
```

(b) Generated checker

Validate Impact of Caught Faults

Transient or tolerable

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

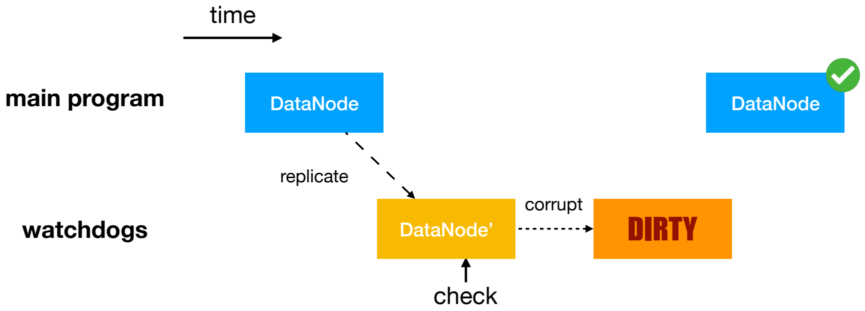
Default: simply re-executes the checker and compare for transient errors

- 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)

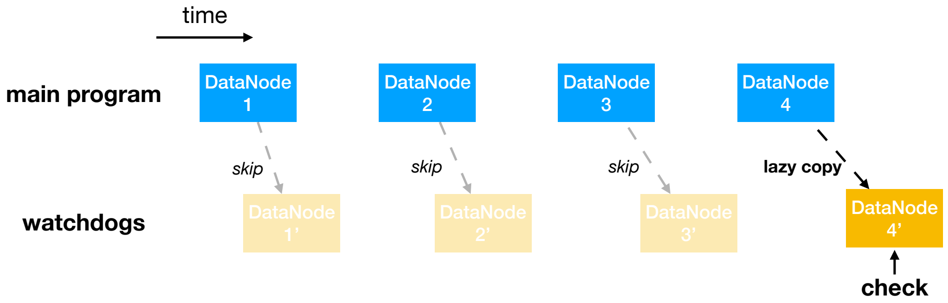
Context manager will replicate the checker context so that any modifications are contained in the watchdog's state



Prevent Side Effects

Context Replication (memory isolation)

To reduce performance overhead: immutability analysis + lazy copy



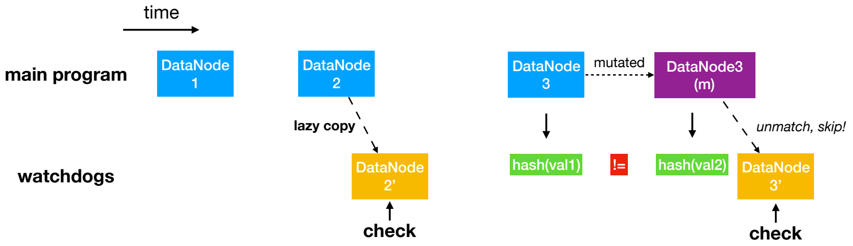
Prevent Side Effects

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`

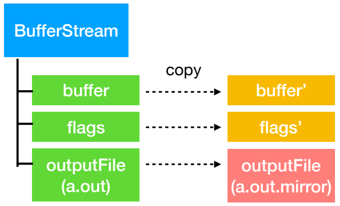


Prevent Side Effects

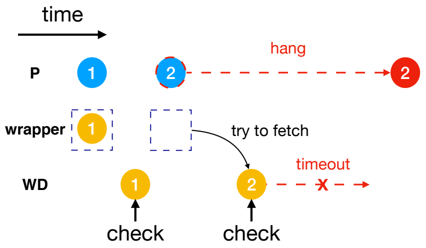
I/O Redirection and Idempotent Wrappers (I/O isolation)

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