Understanding, Detecting and Localizing Partial Failures in Large System Software

By Chang Lou, Peng Huang, and Scott Smith

Presented by Andrey Satarin, @asatarin
May, 2022

https://asatarin.github.io/talks/2022-05-understanding-partial-failures/

Outline

- Understanding Partial Failures
- Catching Partial Failures with Watchdogs
- Generating Watchdogs with OmegaGen
- Evaluation
- Conclusions

Understanding Partial Failures

Partial Failure

A partial failure — a failure in a process P when a fault **does not crash P**, but causes safety or liveness violation or severe slowness for **some functionality**

- It's **process level**, not node level
- Process is still **alive**, this is not a fail-stop failure
- Could be **missed** by usual health checks
- Can lead to **catastrophic outage**

Failure Hierarchy

Failure Hierarchy

Questions

- How do partial failures manifest in **modern systems**?
- How to systematically **detect and localize** partial failures at runtime?

Software	Lang.	Cases	Ver.s (Range)	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

Table 1: Studied software systems, the partial failure cases, and the unique versions, version and date ranges these cases cover.

Findings 1-2

Finding 1: In all the five systems, partial failures appear throughout release history (Table 1). 54% of them occur in the **most recent three years'** software releases.

Finding 2: The root causes of studied **failures are diverse**. The top three (total 48%) root cause types are **uncaught errors**, **indefinite blocking**, and **buggy error handling**.

Findings 3-5

Finding 3: Nearly half (48%) of the partial failures cause some **functionality to be stuck**.

Liveness violations are straightforward to detect

Finding 4: In 13% of the studied cases, a module became a “zombie” with **undefined failure semantics**.

Finding 5: 15% of the partial failures **are silent** (including data loss, corruption, inconsistency, and wrong results).

Findings 6-7

Finding 6: 71% of the failures are triggered by some **specific environment condition**, input, or faults in other processes.

Hard to expose with testing => **need runtime checking**

Finding 7: The majority (68%) of the **failures are “sticky”** — the process will not recover from the faults by itself.

Catching Partial Failures with Watchdogs

Current Checkers

- **Probe** checkers
  - Execute external API to detect issues
- **Signal** checkers
  - Monitor health indicator provided by the system

Issues with Current Checkers

- **Probe** checkers
  - **Large API surface** can’t be covered with probes
  - Partial failures might not be observable at the API level
- **Signal** checkers
  - Susceptible to environment **noise**
  - **Poor accuracy**

Mimic Checkers

- **Mimic-style checkers** — selects some representative operations from each module of the main program, imitates them, and detects errors
- Can pinpoint the faulty module and failing instructions

Intrinsic Watchdog

- Synchronizes state with the main program via **hooks** in the program
- Executes concurrently with the main program
- Lives in the same address space as the main program
- **Generated automatically**

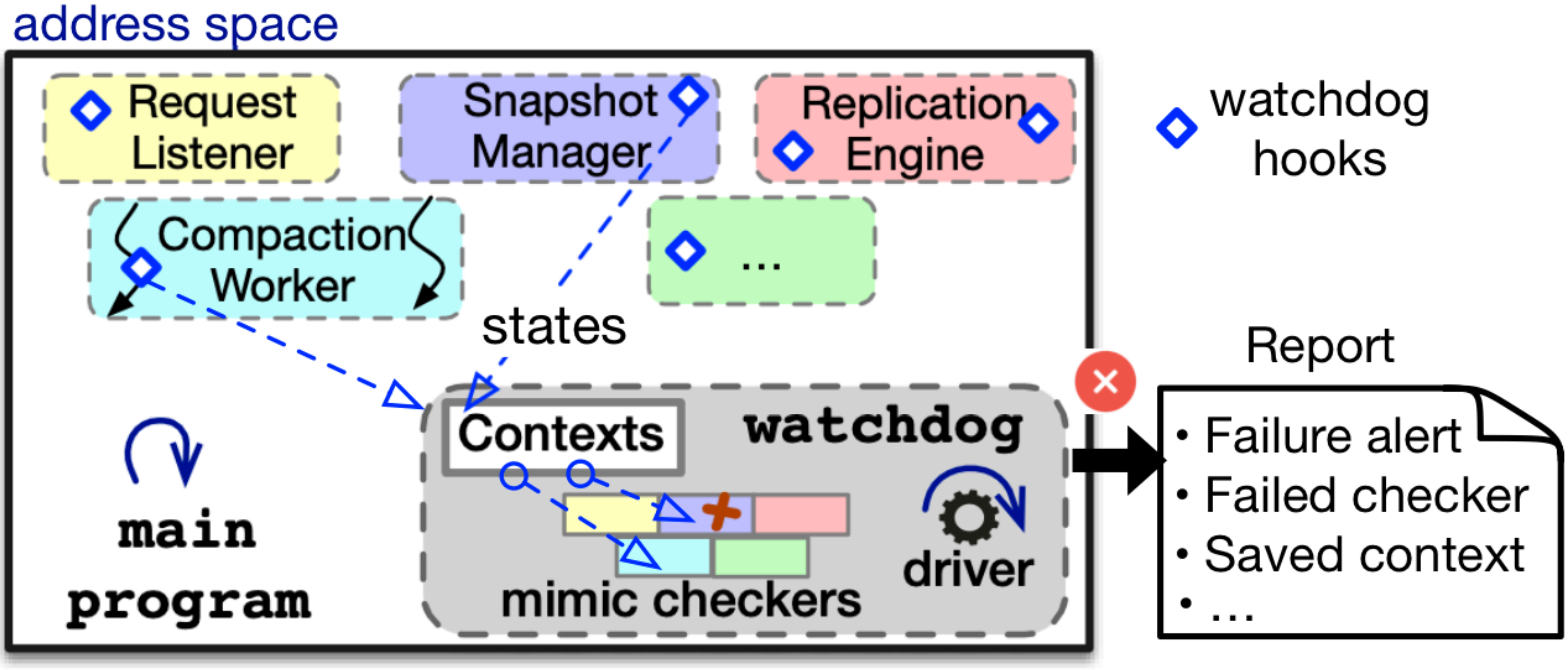

Figure 4: An intrinsic watchdog example.

Generating Watchdogs with OmegaGen

Generating Watchdogs

- Identify long-running methods (1)
- Locate vulnerable operations (2)
- Reduce main program (3)
- Encapsulate reduced program with context factory and hooks (4)
- Add checks to catch faults (5)

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

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

1 identify long-running region

2 locate vulnerable operations

3 reduce

3 reduce

4 insert context hooks

(a) A module in main program

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

4 generate context factory

5 add fault signal checks

(b) Generated checker

Figure 5: Example of watchdog checker OmegaGen generated for a module in ZooKeeper.

Validate Impact of Caught Faults

- Runs validation step to **reduce false alarms**
- Default validation is to **re-run the check**
- Supports manually written validation

Preventing Side Effects

- **Redirect** I/O for writes
- **Idempotent** wrappers for reads
- Re-write socket operations as ping
- If I/O to a another large system => better to apply OmegaGen on that system

Evaluation

Questions

- Does our approach work for **large software**?
- Can the generated watchdogs **detect and localize** diverse forms of real-world partial failures?
- Do the watchdogs provide **strong isolation**?
- Do the watchdogs report **false alarms**?
- What is the runtime **overhead** to the main program?

Detection

- Collected and reproduced **22 real-world failures** in six systems
- Built-in (baseline) detectors did not detect any partial failures
- **Detected 20 out of 22 partial failures** with the median **detection time of 5 seconds**
- Highly effective against **liveness issues** — deadlocks, indefinite blocking
- Effective against **explicit safety issues** — exceptions, errors

Localization

- **Directly pinpoint** the faulty instruction for **55% (11/20)** of the detected cases
- For 35% (7/20) of detected cases, either localize to some program point within the **same function** or some **function along the call chain**
- Probe or signal detectors can only pinpoint the faulty process

False Alarms

- The false alarm ratio is calculated from total false failure reports divided by the total number of check executions.
- The watchdogs and baseline detectors are all configured to **run checks every second**
- Can false alarm ratio be traded for detection time? (Median detection time is 5 seconds)

	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.

	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.

Conclusions

Conclusions

- Study of **100 real-world partial failures** in popular software
- OmegaGen to **generate watchdogs** from code
- Generated watchdogs **detect 20/22** partial failures and **pinpoint scope in 18/20 cases**
- **Exposed new partial failure** in ZooKeeper

The End

Contacts

- Follow me on Twitter @asatarin
- https://www.linkedin.com/in/asatarin/
- https://asatarin.github.io/

References

- Self reference for this talk (slides, video, etc)
- "Understanding, Detecting and Localizing Partial Failures in Large System Software" paper
- Talk at NSDI 2020
- Post from The Morning Paper blog