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

October 15, 2020

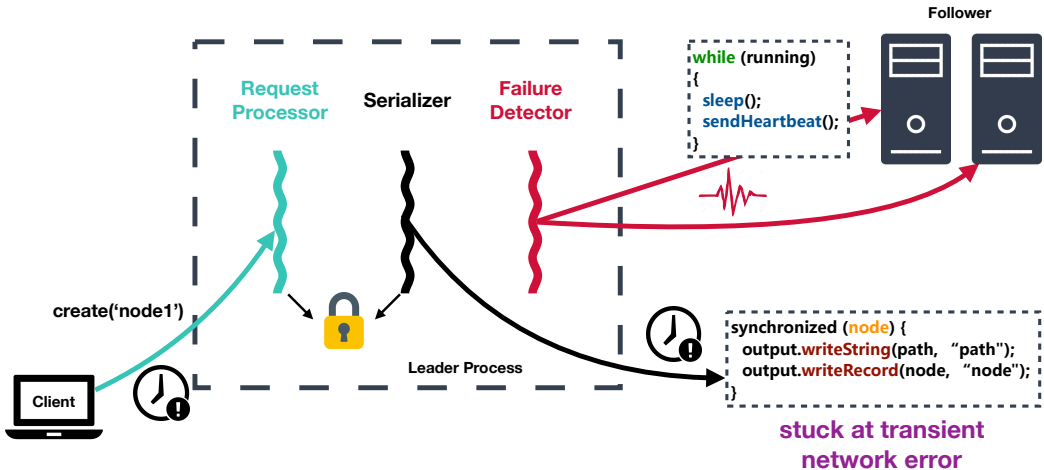
[1] Chang Lou, Peng Huang, and Scott Smith. "Understanding, Detecting and Localizing Partial Failures in Large System Software". In: *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 2020, pp. 559–574.

Overview

- Problem definition
- Case Study
  - Findings
- Motivation
- Proposed Design
  - Ideas

What is a Partial Failure?

An Example

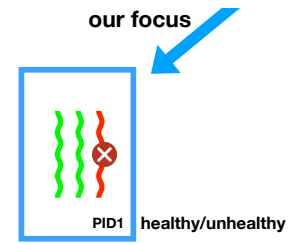


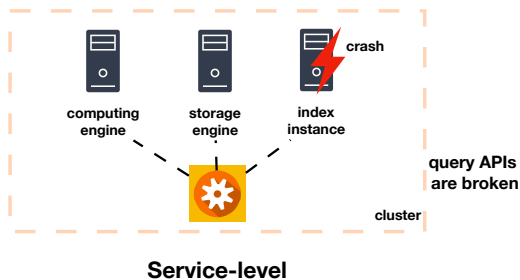
What is a Partial Failure?

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.





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)*

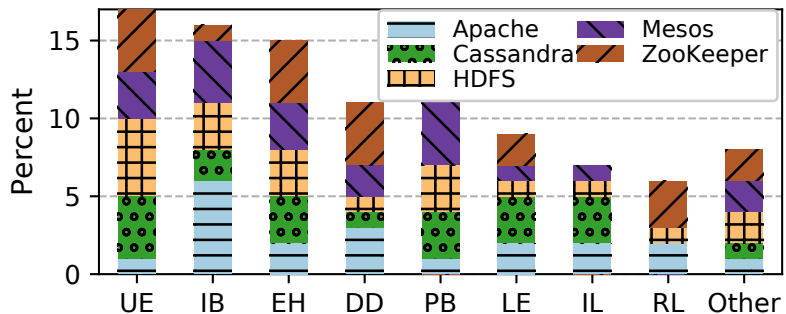
Software	Language	Cases	Versions	Date Range
ZooKeeper	Java	20	17 (3.2.1–3.5.3)	12/01/2009–08/28/2018
Cassandra	Java	20	19 (0.7.4–3.0.13)	04/22/2011–08/31/2017
HDFS	Java	20	14 (0.20.1–3.1.0)	10/29/2009–08/06/2018
Apache	C	20	16 (2.0.40–2.4.29)	08/02/2002–03/20/2018
Mesos	C++	20	11 (0.11.0–1.7.0)	04/08/2013–12/28/2018

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

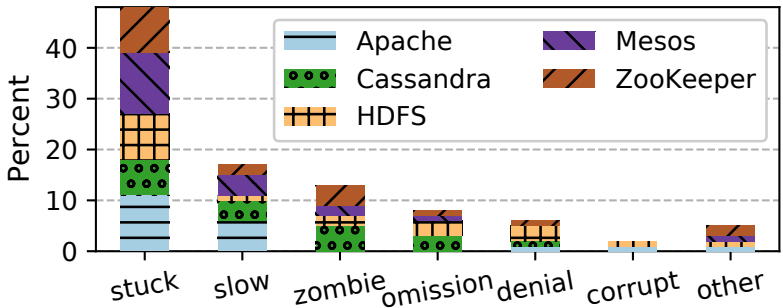


[2]UE: uncaught error; IB: indefinite blocking; EH: buggy error handling; DD: deadlock; PB: performance bug; LE: logic error; IL: infinite loop; RL: resource leak.

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?

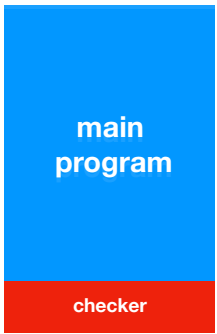
Manual vs generated checkers

Systematically generated checkers to ease developers' burden

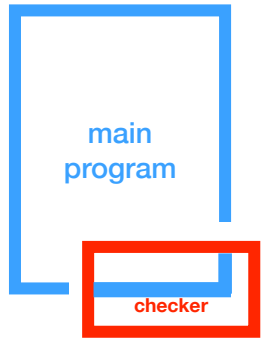
- challenge: difficult to automate for all cases
- opportunity: most of partial failures do not rely on deep semantic understanding to detect, such checkers can potentially be automatically constructed

Intersection Principle

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



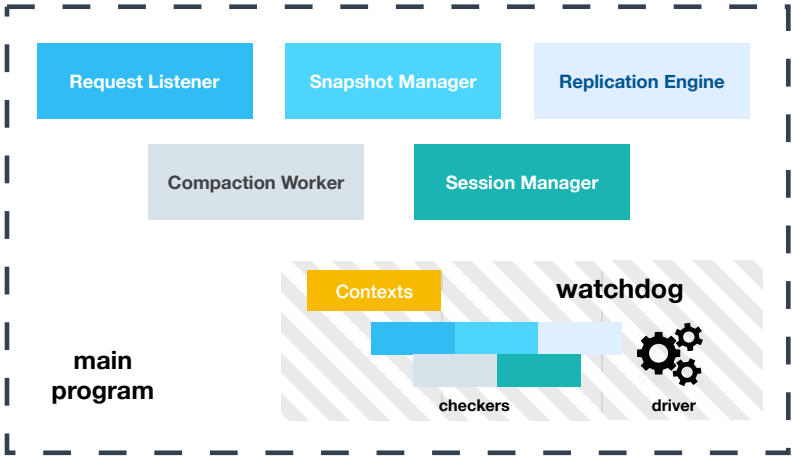
existing approach



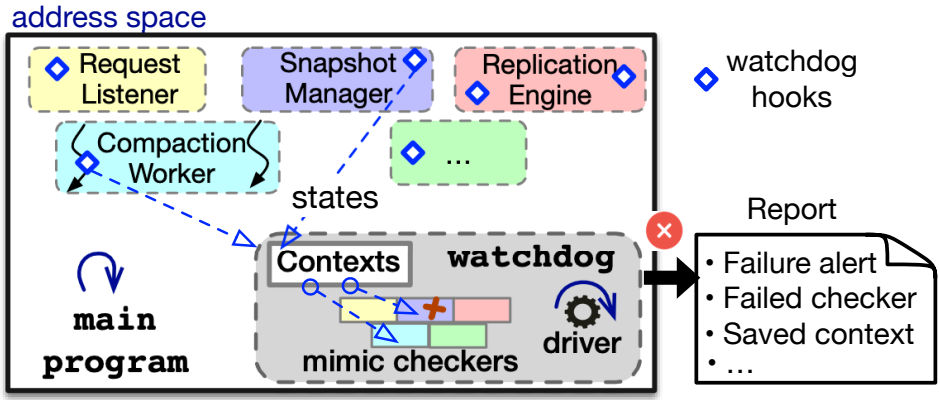
our approach

Intrinsic watchdog: Runtime

An intrinsic watchdog is a dedicated monitoring extension for a process

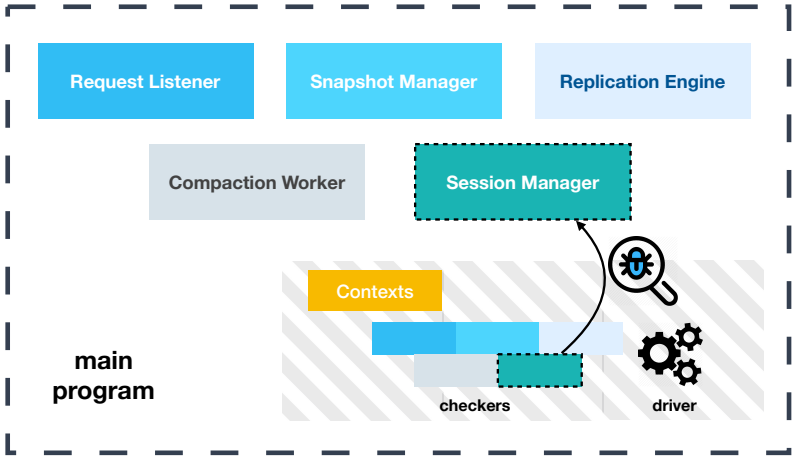


Intrinsic watchdog: How it works?



Characteristic I: Customized

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

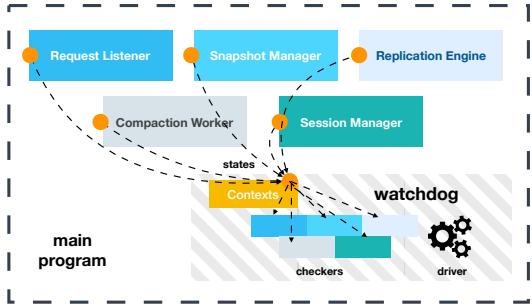


Characteristic II: Stateful

To synchronized states, introduce

Context

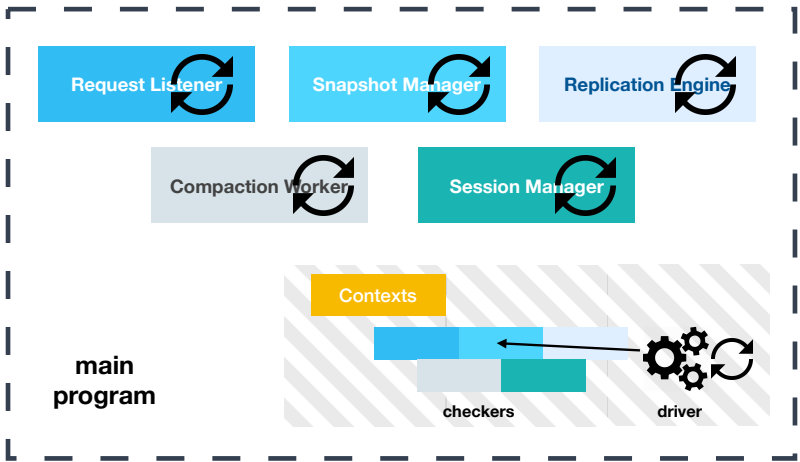
- bound to each checker
- holds all the arguments needed for the checker execution
- synchronized with the program state through hooks in the main program
- update with current state when hooks reached



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