Replicating "Capturing and Enhancing In Situ System Observability for Failure Detection"

Enze Liu
University of California, San Diego
e7liu@eng.ucsd.edu

ABSTRACT

Detecting failures in real world distributed systems is known to be hard. Such systems could fail for various reasons. Despite a rich literature in failure detection, many types of failures, especially gray failure, remain undiscovered and less well understood. A recent paper, titled "Capturing and Enhancing In Situ System Observability", made an attempt to tackle this problem. The authors proposed a system called panorama that improved *system observability* by exploiting the interactions between system components. The advantage of increased system observability is notable: new kinds of failures, such as gray failure, could now be captured; the efficiency and efficacy of failure detection are boosted; localizing the failure has also become easier. Another recent paper following this line of work even received a best paper award at NSDI '20. Drawn by this novel idea and the recent acknowledgment it received from the research community, we are interested in reproducing the original work with the hope that we would be able to master the idea and fully appreciate its value. Besides recreating the original work, we might try evaluating the work in a broader context if time permits.

1 BACKGROUND AND MOTIVATION

Modern distributed systems usually consist of numerous components and have high complexity. As a result, properly and promptly capturing failures in production environments is perceived as an annoying task [2, 4]. While a rich literature has researched the problem of failure detection, it remains challenging. A recent paper at OSDI '18 [3], titled "Capturing and Enhancing In Situ System Observability" by Huang et al., sought to provide a solution to this problem. The authors designed Panorama, a system for detecting production failures in distributed systems. They leveraged the key insight that **system observability could be improved by making each component an observer of other components with which it interacts**. Each observer would provide observations on the components it interacts with. Taking advantage of these first-hand observations, the authors came up with a simple detection algorithm that achieved high detection accuracy. Their evaluation on 15 real-world failures suggested that Panorama were able to detect and localize failures in a fast manner, whereas existing detectors would only detect a fraction of the failures and run much longer. A follow-up work by the same group of researchers received a best paper award at NSDI '20, demonstrating the power of this idea [5].

2 TEAM MEMBERS

Enze Liu will the only one on this team.

3 PRELIMINARY PROGRESS

We have made some initial progress setting up the testing environment and reproducing one test case to ensure that our end goal is achievable. First, we contacted Ryan, who is the lead author of panorama, and received practical advice from them. Following Ryan's suggestions, we succeeded in reproducing one important but easy to set up bug in the Zookeeper [1]. We now are fairly confident that we would be able to have real cases to evaluate. We detail our progress below.

3.1 Acquiring Test Cases Used in the Paper

On Thursday, I sent an email to Ryan Huang at JHU asking for test cases he used in the paper. Fortunately, Ryan was kind enough to respond to my email and attach a few simple test cases he used in the paper. ZOOKEEPER-2201 is one of them [1]. It is a subtle gray failure bug found in Zookeeper version 3.4.6 and 3.5.0. Ryan indicated that this bug might be one of the easiest bugs to reproduce. Thus, we worked on this one first.

3.2 Setting Up Zookeeper Ensemble

We set up a three-server Zookeeper ensemble on the EC2 VM provided to us. First, we downloaded the source code of Zookeeper version 3.4.6 and made a customized version of it. Next, we deployed three Zookeeper servers listening on different ports, forming a Zookeeper ensemble.

3.3 Reproducing ZOOKEEPER-2201

Reproducing ZOOKEEPER-2201 in a real-world setting is nontrivial. We have two issues there. First, we would face great difficulty simulating a real-world scenario, which requires a 20-node cluster plus GBs of data. Second, Ryan noted in his correspondence with us that faults typically need to be injected at careful timing in a real environment. Neither of these two issues could be resolved in a tractable amount of time.

That being said, there is one thing we could do here. Our insight here is that we do not have to be in a real-world situation to trigger ZOOKEEPER-2201. ZOOKEEPER-2201 is reported and fixed in 2015. Along with fixing the bug, the maintainers also attached a test case that examined this specific bug. In that test case, they intentionally introduced a buggy function that would trigger ZOOKEEPER-2201 if it were not fixed. This is convenient for us: we can just use this function to trigger ZOOKEEPER-2201. As we previously noted, we modified the Zookeeper source code and injected this function. We could control when this function is called, and thus controlling when this bug is triggered.

3.4 Future Plan on Configuring Test Cases

As I am now working alone on this project, I am envisioning that I would only test my code on one or two bugs in Zookeeper, which I have already setup on my EC2 VM.

However, being able to trigger a bug is not good enough. We want to make sure that Panorama could detect it and use Panorama as an oracle to test against my implementation of Panorama. To achieve this, two issues need to be addressed. On the one hand, obviously, we need to get Panorama server running. On the other hand, we need to inject code into Zookeeper server so that it could report anything it observes to Panorama server. These two issues are the prerequisite for Panorama to be able to detect any bug. I plan to work on these two issues in the upcoming week.

4 GOALS AND TIMELINE

My main interest is in reproducing the panorama system implemented by the original paper, in which the authors reported an implementation of 6,000 lines of GO code. I envision this part alone nontrivial amount of work. As a first step, **I would aim at delivering the key insight of this work and making it functionable to collect and analysis error in a distributed system, with a projected amount of 2k to 3k lines of GO code**. If I could finish this part fast enough, I would consider evaluating the work in a richer context or even extending the work. Here I divide my goals into two categories: *Must-have* and *Might-have*. I would make sure to deliver the items in *Must-have* and might deliver those in *Might-have* depending on the time available. Below I detail my goals.

4.1 Panorama System

The panorama project consists of two parts: the panorama system and a code injection tool that automatically inserts code based on offline static analysis. The former is implemented in GO and the latter is implemented on top of Soot and AspectJ. As I previously noted, reproducing panorama system itself is nontrivial, and my interest is in distributed systems not static analysis. Thus, I do not consider recreating the code injection tool high priority. If I end up with some free time, I might consider extending panorama with more functionalities. Below, I summarize my goals as in Table 1 (assuming we have 4 weeks of time).

Goals	Type	Estimated LOC	Timeline
Key part of Panorama	***Must-have***	2k - 3k	Week2/3
The rest of Panorama	*Might-have*	2k - 3k	Week4
Code injection tool	*Might-have*	unknown	Week4

Table 1: Goals with regard to implementing the Panorama System assuming we have 4 weeks of time.

4.2 Test Environment and Cases

The authors setup the experiments in a real-world setting: a cluster of 20 physical machine, each with a 2.4 GHz CPU, 64 GB of RAM, and a 480 GB SATA SSD. They then evaluated Panorama with four widely-used distributed systems: ZooKeeper, Hadoop, HBase, and Cassandra. Clearly, these are settings I would not able to achieve. To make my project tractable, **I would not reproduce any bugs in a real-world workflow, instead I would introduce them in an artificial and simple way**, as detailed in section 3.3. Below I detail my goals and timeline for setting up test environment and cases.

Goals	Type	Estimated LOC	Timeline
Reproduce one bug	***Must-have***	N/A	Done
Run Panorama	***Must-have***	N/A	Week1
Find the bug in Panorama	***Must-have***	N/A	Week1
Test my system with the bug	***Must-have***	N/A	Week4
Add more bugs	*Might-have*	N/A	Week4

Table 2: Goals with regard to setting up the test environment and cases assuming we have 4 weeks of time.

REFERENCES

[1] https://issues.apache.org/jira/browse/ZOOKEEPER-2201. https://issues.apache.org/jira/browse/ZOOKEEPER-2201. (????). (Accessed on 05/17/2020).

[2] Jeff Dean. 2009. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS* 1 (2009).

[3] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 18)*. USENIX Association, Carlsbad, CA, 1–16.

[4] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. 2007. WiDS checker: Combating bugs in distributed systems. (2007).

[5] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 559–574. https://www.usenix.org/conference/nsdi20/presentation/lou