# LittlePanorama: A Lite Version of Panorama

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 recent acknowledgment it received from the research community, we decide to reproduce the original work so that we could master the idea and fully appreciate its value. We present LittlePanorama, a system that implements core modules of Panorama and provides the same functionality. To balance our workload, we reuse classes and types defined in Panorama when possible. We evaluate the performance of LittlePanorama and Panorama against bugs used in the original paper.

## 1 INTRODUCTION

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 [6, 9]. While a rich literature has researched the problem of failure detection, it remains challenging.

In fact, many would perceive complete stoppable as the definition of a failure. However, a variety of different kinds of failures exist in production systems. While simple failures could be detected and removed efficiently through extensive testing, others, for example gray failure, remain undiscovered due to their obscure and elusive nature. Gray failure is the type of failure where a system would appear to be alive and functioning but is actually experiencing issues with certain components [8]. For instance, a background thread responsible for writing data to disk could get stuck while other threads are still functioning. Other examples include non-critical component failure, performance degradation, random packet loss, flaky I/O, memory thrashing and capacity pressure [8]. Often times, the larger your scale, the more common gray failures become [1]. While some gray failures only have moderate impact on the system, other may result in a catastrophic. An example is ZooKeeper-2201 [2], a gray failure found in ZooKeeper version 3.4.6 and 3.5.0. It causes the cluster to hang for over 15 minutes even though the leader would still exchange heartbeat messages with its followers [5].

Current state-of-the-art approaches struggle at detecting such failures. A recent paper at OSDI '18 [7], titled "Capturing and Enhancing In Situ System Observability" by Huang et al., sought to provide a new solution. The observation here is that modern distributed systems generally involve many highly interactive components. When one fails, it is very likely that others would be able to observe it through their interaction with the failed one. So one could totally monitor the status of a component through other components that interact with it. Additionally, different components could report on the status of the target component from different dimensions, including but not limited to memory, cpu and network status of the target. The authors find that **turning each component into an observer of other components with which it interacts greatly improves a system's observability**.

Inspired by this finding, the authors design and implement Panorama, a generic failure detection framework that leverages and enhances system observability to detect complex production failures [7]. It extracts essential observations from various components and makes inference on the status of a component based on them. Also, given the fact that the majority of components in modern systems are programmed to only handle the errors they encounter when interacting with a failed component but not report them, the authors build a tool that automatically turns components into observers that are capable of reporting observations.

They evaluate Panorama by applying it to detect failures in real-world software like ZooKeeper, Cassandra, HBase and HDFS. They find that it significantly outperforms other state-of-the-art failure detection tools with minor overhead. A follow-up work by the same group of researchers even received a best paper award at NSDI '20 [10]. Attracted by the success of recent work around this idea, we decide to

recreate the original work so that we could have a better understanding of the idea and fully grasp of its value.

We present LittlePanorama, a system that implements core modules of Panorama and provides the same functionality. We build LittlePanorama on top of the classes and types defined in Panorama. We reimplement most modules with much simpler data structures, yet retaining the same program logic and function signatures. One could easily replace Panorama with LittlePanorama, and vice versa. Since we do not have access to the tool that automatically turns components into observers, we do so manually. We demonstrate that LittlePanorama is functioning properly by evaluating it against two artificial crash failures. We then compare LittlePanorama against Panorama using two real-world gray failures. Additionally, we prove that LittlePanorama is able to detect and reflect status changes of a component. Finally, the authors do not disclose any failures used in the paper, nor do they publish anything for reproducibility. We release the two bugs we acquired from the authors and publish all the scripts/software artifacts we build so that other researchers who are also interested Panorama do not have to repeat what we have done.

The rest of this paper is structured as follows. Section 2 introduces the LittlePanorama system. Section 3 discusses our implementation. Section 4 shows how we create failures for testing, while Section 5 evaluates LittlePanorama on those failures. We summarize our thoughts in Section 6, 7 and 8.

## 2 LITTLEPANORAMA SYSTEM

### 2.1 Overview

In LittlePanorama, we implement the core features of Panorama in a simplified manner. One can think of LittlePanorama as the lite version of Panorama. That said, at a high level, the design of LittlePanorama and Panorama are still much alike.

Figure 1 gives an overview of LittlePanorama. In a nutshell, LittlePanorama consists of two parts, LittlePanorama servers and observers. Each observer reports observations to a server on components it interacts with. Servers then make inference on whether one component has failed or not based on the observations received. They would also exchange observations such that all servers end up with the same set of observations. This idea of leveraging the interaction between components for failure detection is not new. Previous work [11] has employed similar ideas, but in a limited scope. By comparison, LittlePanorama (and of course Panorama) allows every component to report observations on any other component it interacts with. This gives LittlePanorama the ability to gather observations from a variety of perspectives.

As a simple example, consider a program 1 in Figure 1. If only threads (in this case A, B and C) within program 1

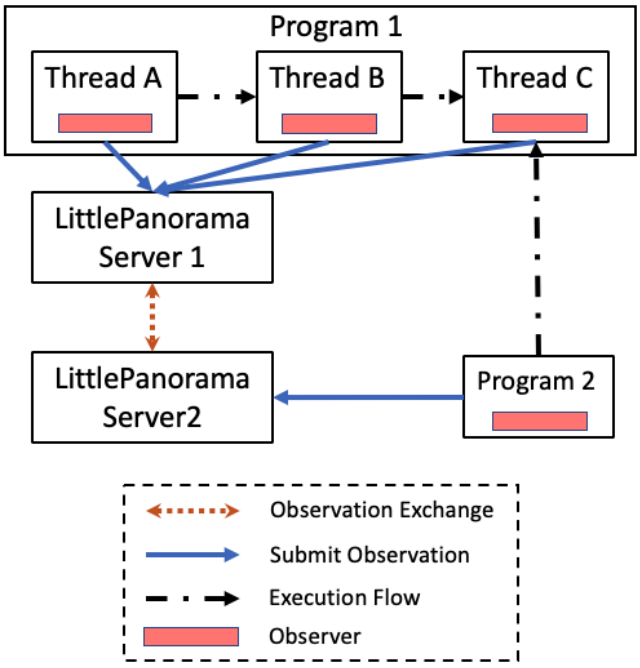


**Figure 1: Overview of LittlePanorama**

could report observations on each other, it is unlikely that one would detect issues like network partition, as everything within program 1 remains functioning. However, if its peers, in this case program 2, could report observations on it, it would be easy to detect that program 1 is suffering from a network partition because program 1 would now become unreachable. Conversely, observers outside program 1 might not be able to report on the program's cpu/memory usage, and thus we would need observers within program 1 to detect if any thread is experiencing failures such as memory leak and cpu thrashing.

Below we briefly discuss the key differences between the design of Panorama and LittlePanorama.

- **Where servers are located.** For each program monitored, a Panorama server instance would be colocated with it as its local storage for observations. In contrast, LittlePanorama servers are not colocated with any monitored program for simplicity. They are independent programs that may or may not run on the same machine as monitored programs.
- **How observers are created.** Both LittlePanorama and Panorama leverages code logic in existing components and make them observers. However, Panorama employs a static tool to automatically do so, while LittlePanorama does not have such a tool. Creating observers in LittlePanorama is done manually. Further details are provided in Section 2.4.

- **Whether localizing failure is supported.** Besides deciding the status of a component, Panorama goes a step further by trying to localize a failure. To achieve this, Panorama introduces *context*, a variable that describes what an observer is doing when it makes an observation. LittlePanorama does not have an implementation for *context*.
- **The inference engine.** The authors report in the original paper that they use a bounded-look-back majority algorithm in their inference engine. We do not have a bound on the number of look-backs performed. More details are given in Section 2.6.

## 2.2 Abstractions and APIs

One goal of LittlePanorama is to maintain the same abstraction and APIs as Panorama. Having the same APIs largely reduces the effort needed to reuse existing code from Panorama. In this section, we present terms and abstractions defined in the original paper [7] with minor modification.

Table 1 lists the abstractions and terms that are used in both LittlePanorama and Panorama. A component is either a thread or a process. A component is an observer if it is capable of making observations on other components. Those that being observed are called subjects. In many cases, a component would be both an observer and a subject. Observations are sent to servers and stored there. Status defines the how healthy a component is and can only be one of eight predetermined values, including HEALTHY and UNHEALTHY. One special value is PENDING, the use of which is beyond the scope of this project. More details can be found in the original paper. An observation contains information on a subject, which includes the identity of the observer and the subject, a timestamp and the observed status of the subject.

Servers define a set of unified APIs for communicating with each other and observers. An example is "SubmitObservation", which is called by an observer to submit an observation. Other examples include "RegisterObserver", "GetInference" and "ExchangeObservation". Additional data structures and APIs are maintained by Panorama for enhanced functionalities. We consider them out-of-scope for this project.

## 2.3 LittlePanorama Server

LittlePanorama servers run independently as separate programs, while in Panorama they are colocated with monitored programs. Each LittlePanorama server has three key responsibilities: store all the observations sent to it, disseminate observations to other servers and make inference on the status of a subject using the inference engine described in Section 2.6. It maintains an in-memory data structure map that saves all the observations made on each subject. The map is keyed by subject so that observations for a subject could be indexed efficiently. Unlike Panorama, we do not save anything to disk, which could be easily done with some engineering effort if we were to do it. Other differences between LittlePanorama and Panorama include whether to save all the observations or only n observations, whether a GC is implemented and whether to precompute an inference for each subject. We consider these differences to be of little importance.

| | |
|---|---|
| **Component** | a process or thread |
| **Subject** | a component to be monitored |
| **Observer** | a component monitoring a subject |
| **Server** | Panorama or LittlePanorama instance that receives observations and/or makes verdict |
| **Status** | the health situation of a subject |
| **Observation** | information an observer collects on a subject's status |
| **Inference** | a decision about a subject's status by summarizing a set of observations |

**Table 1: Abstractions and terms used in LittlePanorama**

## 2.4 Observers

One cool thing about Panorama is that it does not use any dedicated detectors. Instead, it turns existing components into observers by injecting API hooks into components' source code while keeping the functionalities unchanged. However, whenever an observer encounters an exception, it now not only handles it but also sends an observation through the API hooks. The process of adding hooks is done automatically in Panorama using a tool.

Unfortunately, this tool is not published by the authors. We have to manually inject hooks. These hooks are generally injected at the boundaries between different components. In other words, hooks are expected to be injected in functions that switch between component. The authors give a brief guide on how to find these functions. It is still a challenging task without enough domain knowledge and time. In this project, we manually inject hooks to a few components without guaranteeing that they are injected at boundaries.

That said, this does not mean that we are injecting hooks in the wrong way or in the wrong place. In fact, we inject hooks at near-boundaries. For example, ZooKeeper leader would ping its peers regularly. The boundary in this case would be the function that makes the RPC call and sends the ping packet to peers. Locating this function in thousands lines of code is nontrivial without enough domain knowledge. However, we could easily identify the function that initiates

the ping request. We would then inject a hook into that function.

## 2.5 Observation Exchange

In both Panorama and LittlePanorama, different observers might submit their observations on the same subject to different servers. To make sure that all servers see the same set of observations and thus making the same inference, a propagation protocol is used. Servers leverage this protocol to propagate/receive observations to/from each other. Besides this protocol, Panorama supports more sophisticated operations such as subscribing/unsubscribing to a subject. This is not implemented in LittlePanorama. Also, both Panorama and LittlePanorama assume a perfect network. Dealing with network failure is another research question of its own.

## 2.6 Inference Engine

After collecting all the observations, an inference engine is used to decide the current status of a subject. The authors report in the original paper that they are using a bounded-look-back majority algorithm. When we implement it, we do not puts a bound on the number of look-backs performed. Our algorithm works as follows. 1. observations are grouped by observer. 2. for each group, observations are sorted in reverse chronological order. 3. we pick the status reported in the latest observation as the inference from this group. 4. we inspect observations from the latest to the oldest and average the scores reported in each observation. We stop when we hit an observation that reports a different status. 5. the status of a subject is determined via a majority vote across all the groups. Again, the authors' algorithm put a bound in step 4 while we do not.

# 3 IMPLEMENTATION

We build LittlePanorama on top of classes and types defined in Panorama. We keep the same program interfaces when possible. Panorama defines and implements most classes, types and interfaces using protocol buffers [3]. Reusing its code largely reduces our workload so that we do not have to implement everything from scratch. Retaining the same set of interfaces allows easy alternation between Panorama and LittlePanorama.

We implement LittlePanorama server in ~1, 200 lines of Go code. This number excludes code imported from Panorama, modification made to ZooKeeper's Java source code and other scripts written in bash. Additionally, as we previous mentioned, failures used to evaluated Panorama are not disclosed. In contrast, we publish the two gray failures we acquired from the authors. Details regarding these two failures are in Section 4. Further, the authors evaluate these failures in real-world settings, making it hard to reproduce their results. On the contrary, as we detail later in Section 4, we modify ZooKeeper's source code and inject these failures in a simple way such that it can be easily reproduced. To encourage further scientific exploration, we are open-sourcing our implementation of LittlePanorama as well as our customized version of ZooKeeper [1].

| Module | Description | Imported |
|---|---|---|
| build | generated by protocol buffers | Yes |
| cmd | main function | No |
| decision | decision engine | No |
| exchange | propagation protocol | No |
| service | type/class defs | Yes |
| store | data storage | No |
| type | additional types | No |
| util | utility functions | No |
| vendor | other third-party modules | Yes |

**Table 2: Modules included in LittlePanorama and whether they are imported from Panorama.**

Since we reuse code from Panorama, it is also important to distinguish code written by us and code imported. Table 2 lists modules that are included in LittlePanorama and whether they are imported from Panorama.

# 4 CREATING FAILURES

A key prerequisite for evaluation is to create failures so that LittlePanorama could be used to detect them. There are two possible ways to do it. One is to inject artificial bugs that create certain failures. For example, we could inject some buggy code that would crash a system. The other is to leverage existing bugs in the system. As an example, consider a program that dereference a pointer without checking whether it is null or not. We could trigger an exception by either crafting an input that sets the pointer to null or running a production workflow with the hope that the pointer would be set to null. The former approach is easy to achieve while the latter models real-world scenarios better. The latter is also orders of magnitude harder. First, one needs to identify bugs in existing systems. Second, one must be able to trigger those bugs, ideally in a production workflow and in a deterministic fashion. Finally, our project adds another layer of complexity here: we want the bugs to cause gray failures.

To keep our project in a manageable scope, we create failures in a hybrid manner. We first inject two artificial crash failures for ZooKeeper. For gray failures, we want to leverage existing gray failures instead of creating our own. We are also in favor of bugs that are representative and easy

[1] https://github.com/alexliu0809/LittlePanorama

| BugId | Description | Gray |
|---|---|---|
| artificial-01 | An artificially injected bug that causes the leader to fail | No |
| artificial-02 | An artificially injected bug that causes the follower to fail | No |
| zookeeper-2201 | Zookeeper service becomes unavailable due to transient network partition | Yes |
| zookeeper-2247 | Zookeeper service becomes unavailable when leader fails to write transaction log | Yes |

**Table 3: Bugs used in our experiments. Gray indicates whether the resultant failure is a gray failure or not.**

to create and test. Sadly, the original Panorama paper do not have references to the bugs they used. Thus, the first thing we do is to acquire the list of bugs evaluated in the paper. We contact Ryan, who is the lead author of panorama, and are fortunate to hear back from him. In his correspondence with us, he not only includes links to a subset of bugs used in the paper but also indicates which ones are easy to start with. ZOOKEEPER-2201 [2] is among the easy ones. Since we read the Zookeeper paper in this class, we decided to focus on this one first. After some search on the Internet, we are lucky enough to identify another gray failure bug — ZOOKEEPER 2247 [4]. Both bugs are used in the original paper. In total, we have created four failures, listed in Table 3.

Below we provide more detail on how they are created and triggered.

## 4.1 Artificial Crash Failures

We introduce two artificial crash failures, *artificial-01* and *artificial-02*, which would crash ZooKeeper leader and follower respectively when triggered. As previously stated, injecting and triggering artificial failures are easy. Creating crash failure is even easier. In fact, we do not explicitly inject any code to ZooKeeper. Instead, we simulate the crash failure of a program by terminating it with linux command *kill*. Once a subject is killed, other observers that are interacting with it should observe its failure and report to LittlePanorama server. We verify that this is case later in Section 5.

## 4.2 ZOOKEEPER-2201

*4.2.1 Description.* ZOOKEEPER-2201 [2] is a bug that affects ZooKeeper version 3.4.6 and 3.5.0. It can cause the entire cluster to hang for 15 minutes on Linux. During this time, any attempt to create/delete/modify data will hang. Also, because the ping packets that leader exchanges with followers are sent by another thread which is unaffected, followers never time out and become leader, even though the cluster will make no progress. It is a perfect gray failure bug for us to evaluate.

*4.2.2 Triggering.* In a real-world setting, ZOOKEEPER-2201 can be triggered by injecting a network partition. However, triggering it in a production 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, introducing network partition does not guarantee that ZOOKEEPER-2201 would be triggered. We would need to inject network partition at a careful timing. In fact, it is typically that faults need to be injected at careful timing in a real environment to trigger a failure, as Ryan notes in his correspondence with us. Clearly, neither of these two issues could be resolved in a tractable amount of time. We have to find another solution.

Fortunately, there is one thing we could do. Our insight here is that we do not have to be in a real-world situation to trigger ZOOKEEPER-2201. ZOOKEEPER-2201 was reported and fixed in 2015. Along with fixing the bug, the maintainers 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.

We first add this function to Zookeeper's source code. We then verify that a call to this function would trigger ZOOKEEPER-2201. Finally, we need to determine when and how to call this function. We decide that whenever a client sent a request to create the data at node */gray1*, we would invoke the injected buggy function, and thus triggering ZOOKEEPER-2201.

Now we have successfully triggered ZOOKEEPER-2201 and could also control when it would be triggered.

*4.2.3 Detection.* We are now ready to detect ZOOKEEPER-2201. We first trigger it by asking a client to create data at node */gray1*. After this, the leader will hang and stop responding to clients. However, it would still exchange heartbeat messages with its peers — the other Zookeeper servers. This means its peers would report to our LittlePanorama server that the Leader is healthy.

We now need some help from ZooKeeper clients. Since the leader is hanging, any connection established by a client to the leader will timeout, informing the client that the leader is having some issues. Thus, in this case, a client would observe and report to our LittlePanorama server that the leader is having network issues, while a peer Zookeeper server would report no error on the leader's network connectivity. Also, since our LittlePanorama server uses a majority vote algorithm to determine the status of a subject, we would need at least as many clients as peer ZooKeeper servers. For

example, if there are two peer Zookeeper servers, we need at least two clients to report error on the leader's network connection. This can be done trivially. Finally, we confirm that our LittlePanorama server could detect the network error in Section 5.

## 4.3 ZOOKEEPER-2247

*4.3.1 Description.* ZOOKEEPER-2247 [4] is a bug that affects ZooKeeper version 3.5.0. It makes ZooKeeper service unavailable for a short period of time. During this time, leader has failed, yet still remaining the leader. Because leader has failed, any request from clients trying to create/delete/modify data would fail. Other peer servers would also be able to sense that the leader has failed when exchanging heartbeat with it.

*4.3.2 Triggering.* We use the same approach described in Section 4.2.2. We find the relevant test function that triggers ZOOKEEPER-2247. We add this function to Zookeeper's source code with minor changes. We then verify that a call to this function would trigger ZOOKEEPER-2247. Finally, we decide that whenever a client sent a request to create the data at node */gray2*, we would invoke the injected buggy function, and thus triggering ZOOKEEPER-2247.

Now we have successfully triggered ZOOKEEPER-2247 and could also control when it would be triggered.

*4.3.3 Detection.* Detecting ZOOKEEPER-2247 is much easier than that of ZOOKEEPER-2201. Since the leader has failed, both peers ZooKeeper servers and clients would be able observe its failure. We just need a client to trigger it.

## 4.4 A Bonus Failure

As mentioned in Section 4.2, ZOOKEEPER-2201 would cause the entire cluster to hang for a certain period of time, and then the cluster would get back to normal. Lucky enough, we find a variable that allows us to control how long the cluster would hang. This gives us the ability to test another type of failure: intermittent failure. An example of intermittent failure is when a subject goes from healthy to unhealthy and then come back to healthy. We evaluate LittlePanorama on intermittent failure in Section 5.

# 5 EVALUATION

In this section, we evaluate LittlePanorama to answer the following four questions: (1) How efficient is LittleParorama? (2) Can LittlePanorama detect crash failures? (3) Can LittlePanorama detect gray failures? (4) Can LittlePanorama detect intermittent failures properly?

We evaluate LittlePanorama on failures caused by 4 bugs listed in Table 3. As is mentioned above, the two bugs that crash leaders/followers are introduced artificially for the purpose of demonstration. The other two that result in gray failures are bugs found in production and used in the original paper. We evaluate these bugs in an ideal way — triggering them manually instead of simulating a production workflow. The main constraint that limits us from evaluating LittlePanorama on more bugs or in a more complex setting is the time and effort needed.

| Operation | Panorama | LittlePanorama |
|---|---|---|
| Report | 753 $\mu$s | 194 $\mu$s |

**Table 4: Average speed of reporting an observation (Report)**

## 5.1 Experiment Setup

We use the VM provided to run all the experiments. This VM has a single-core 2.4 GHz Intel Xeon E5-2676 CPU, 1 GB of RAM and a 7.7 GB HVM disk. It runs Ubuntu 18.04 LTS (Bionic Beaver). We also modify the source code of Zookeeper version 3.4.6 to introduce hooks and artificial bugs. We evaluate LittlePanorama with a three-node Zookeeper ensemble listening on different ports.

## 5.2 Performance

*5.2.1 Reporting Speed.* Table 4 shows the average speed of reporting an observation in LittlePanorama and Panorama respectively. Two things can be observed here. First, reporting an observation is cheap, as the average speed is less than 1 ms. This is because reporting an observation generally only incurs a local RPC. Second, LittlePanorama is about 3x faster than Panorama in terms of reporting speed. We attribute LittlePanorama's performance gain to the fact that it implements a much simpler data structure and saves everything in memory.

*5.2.2 Inference Speed.* Figure 2 plots the average inference speed of LittlePanorama and Panorama with varying number of observations. LittlePanorama's performance degrades as the number of observations increases, as in the worst case it would examine all observations. However, Panorama does not have this problem. After a close inspection of Panorama's source code, we realize that this performance stability is due to the fact that Panorama only considers the two most recent observations. The original paper refers to this behavior as *bounded-look-back*.

*5.2.3 Propagation Delay.* Figure 3 graphs the average delay of propagating one observation to all peers with various peer sizes. The propagation delay in both cases is proportional to the number of peers, matching what is reported in the original paper. However, looking at Panorama's source code, it seems that the authors have parallelized the process

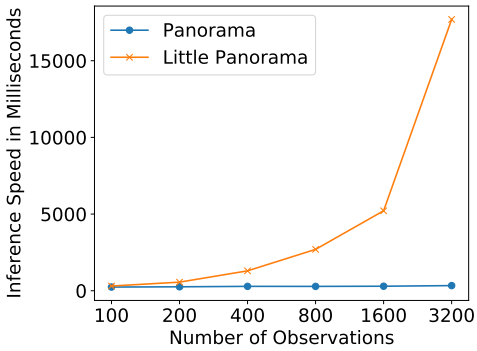


**Figure 2: Inference Speed**

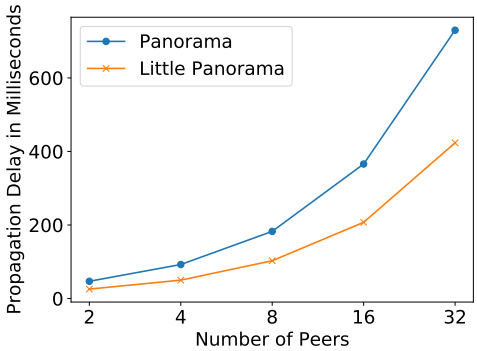


**Figure 3: Propagation Delay**

of propagation, which in theory should give a better performance. Unfortunately, we fail to observe any improvement. Our benchmark code is carefully examined such that the probability of this phenomenon caused by a bug in our code is low. Another possible explanation is that the CPU used in our experiments only has one core, burying the performance gain of thread-level parallelism.

## 5.3 Detection of Crash Failures

Both Panorama and LittlePanorama are designed to detect complex real-world failures. Detecting simple crash failures can serve as a sanity check on the capability of both systems. We inject two crash failures bugs — artificial-01 and artificial-02, which will crash leader and follower respectively when triggered. Table 5 shows the detection time. We see that it takes less than 200 ms to detect both failures, matching the results (~10 ms) reported in the original paper. Further, we observe little difference in performance between LittlePanorama and Panorama.

## 5.4 Detection time for gray failures

Being able to detect complex gray failures is one of the keys goals. To evaluate both systems' ability to detect gray failures, we reuse two gray failures from the original paper. They are

| BugId | Panorama | LittlePanorama |
|---|---|---|
| artificial-01 | 128 ms | 138 ms |
| artificial-02 | 81 ms | 45 ms |

**Table 5: Detection time for crash failures**

| BugId | Panorama | LittlePanorama |
|---|---|---|
| zookeeper-2201 | 66850 ms | 67038 ms |
| zookeeper-2247 | 2632 ms | 2195 ms |

**Table 6: Detection time for gray failures**

real-world production failures found in ZooKeeper. Both of them would cause the ZooKeeper service to be temporarily unavailable but not crashing any ZooKeeper server. Table 6 shows the detection time for both failures using both systems. The detection time for ZOOKEEPER-2201 and ZOOKEEPER-2247 is about 67 seconds and 2 seconds, respectively. At a first glance, one might think 67 seconds is too long and unacceptable. In fact, this is quite expected. As is mentioned earlier in Section 4.2.3, detecting ZOOKEEPER-2201 relies on connections initiated by clients timeout on the leader. The default timeout value is 30 seconds. Observing timeout from two clients would take ~60 seconds. In contrast, for ZOOKEEPER-2247, all observers interacting with the leader would be able to sense its failure soon after it fails, resulting in a much shorter detection time.

## 5.5 Intermittent Failures

Finally, we test LittlePanorama's ability to detect intermittent failures using the bonus failure described in Section 4.4. We instrument the cluster to hang for 2 minutes.

Figure 4 plots the result. The blue line at the bottom shows the actual status of the leader during the timespan of 130 seconds. This line serves as the ground truth. The orange line and green line represent the status of the leader inferred by LittlePanorama and Panorama, respectively. Both of them detect that the leader goes back to normal at time ~125 seconds. It is clear that they not only are able to detect failures but also could monitor status changes in intermittent failures.

# 6 DISCUSSION

The Panorama system essentially consists of two parts: Panorama servers and observers. We reimplement the core functions of Panorama servers and do not encounter any significant obstacles in that process. The design and program logic of Panorama seems straightforward, yet very powerful. However, the process of injecting API hooks into ZooKeeper

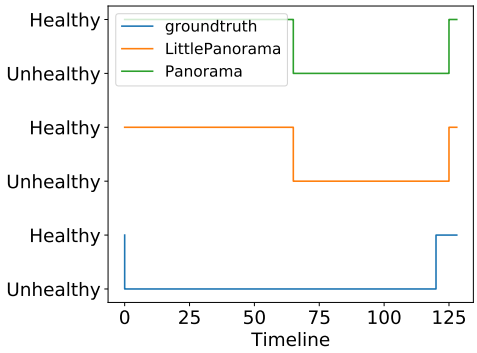


**Figure 4: Detection Time and Inferred Status for Intermittent Failure**

is less pleasant. We are not sure why the authors do not publish the tool that injects API hooks automatically. We realize that this tool is actually of great importance, as where the hooks are injected has a big impact on the detection speed. Injecting hooks at critical places could potentially boost bug detection, while injecting recklessly would result in nothing but overhead. Sadly, the authors do not elaborate too much on how they insert hooks, besides a few short sections. Another thing that the authors could have done is to disclose the bugs that they use in their evaluation, even though readers might not be able to reproduce them given their complexity.

## 7 CONCLUSION

In summary, in this work, we present LittlePanorama, a system that implements core modules of Panorama and provides the same functionality. We build LittlePanorama on top of the classes and types defined in Panorama. We reimplement most modules with much simpler data structures, yet retaining the same program logic and function signatures. We first demonstrate that LittlePanorama is functioning properly by evaluating it against two artificial crash failures. We then compare LittlePanorama against Panorama using two real-world gray failures that are also used by the authors. Additionally, we prove that LittlePanorama is able to detect and reflect status changes of a component. Finally, we release the two bugs we acquire from the authors and publish all the scripts/software artifacts we build for other researchers who might also be interested in Panorama.

## 8 ACKNOWLEDGEMENT

I would like to thank Peng (Ryan) Huang for being responsive. I also thank my advisors Stefan and Geoff for being supportive and understanding, otherwise I would not have been able to finished this project.

## REFERENCES


[1] Gray failure: the AchillesâĂŹ heel of cloud-scale systems âĂŞ the morning paper. (????). https://blog.acolyer.org/2017/06/15/gray-failure-the-achilles-heel-of-cloud-scale-systems/

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

[3] Protocol Buffers Âă|Âă Google Developers. (????). https://developers.google.com/protocol-buffers/

[4] [ZOOKEEPER-2247] Zookeeper service becomes unavailable when leader fails to write transaction log - ASF JIRA. (????). https://issues.apache.org/jira/browse/ZOOKEEPER-2247

[5] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. 2016. Debugging distributed systems. *Queue* 14, 2 (2016), 91–110.

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

[7] 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.

[8] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 150–155.

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

[10] 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

[11] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. 1998. A gossip-style failure detection service. In *MiddlewareâĂŹ98*. Springer, 55–70.