Capturing and Enhancing *In Situ* System Observability for Failure Detection

Peng Huang, *Johns Hopkins University;* Chuanxiong Guo, *ByteDance Inc.;*
Jacob R. Lorch and Lidong Zhou, *Microsoft Research;* Yingnong Dang, *Microsoft*

https://www.usenix.org/conference/osdi18/presentation/huang

This paper is included in the Proceedings of the
13th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the
13th USENIX Symposium on Operating Systems
Design and Implementation
is sponsored by USENIX.

Capturing and Enhancing *In Situ* System Observability for Failure Detection

Peng Huang
Johns Hopkins University

Chuanxiong Guo
ByteDance Inc.

Jacob R. Lorch
Lidong Zhou
Microsoft Research

Yingnong Dang
Microsoft

Abstract

Real-world distributed systems suffer unavailability due to various types of failure. But, despite enormous effort, many failures, especially gray failures, still escape detection. In this paper, we argue that the missing piece in failure detection is detecting what the requesters of a failing component see. This insight leads us to the design and implementation of Panorama, a system designed to enhance *system observability* by taking advantage of the interactions between a system's components. By providing a systematic channel and analysis tool, Panorama turns a component into a logical observer so that it not only handles errors, but also *reports* them. Furthermore, Panorama incorporates techniques for making such observations even when indirection exists between components. Panorama can easily integrate with popular distributed systems and detect all 15 *real-world* gray failures that we reproduced in less than 7 s, whereas existing approaches detect only one of them in under 300 s.

1 Introduction

Modern cloud systems frequently involve numerous components and massive complexity, so failures are common in production environments [17, 18, 22]. Detecting failures reliably and rapidly is thus critical to achieving high availability. While the problem of failure detection has been extensively studied [8, 13, 14, 20, 24, 29, 33, 34, 47], it remains challenging for practitioners. Indeed, system complexity often makes it hard to answer the core question of *what constitutes a failure*.

A simple answer, as used by most existing detection mechanisms, is to define failure as complete stoppage (crash failure). But, failures in production systems can be obscure and complex, in part because many simple failures can be eliminated through testing [49] or gradual roll-out. A component in production may experience gray failure [30], a failure whose manifestation is subtle and difficult to detect. For example, a critical thread of a process might get stuck while its other threads including a failure detector keep running. Or, a component might experience limplock [19], random packet loss [26], fail-slow hardware [11, 25], silent hanging, or state corruption. Such complex failures are the culprits of many real-world production service outages [1, 3, 4, 6, 10, 23, 30, 36, 38].

As an example, ZooKeeper [31] is a widely-used system that provides highly reliable distributed coordination. The system is designed to tolerate leader or follower crashes. Nevertheless, in one production deployment [39], an entire cluster went into a near-freeze status (i.e., clients were unable to write data) even though the leader was still actively exchanging heartbeat messages with its followers. That incident was triggered by a transient network issue in the leader and a software defect that performs blocking I/Os in a critical section.

Therefore, practitioners suggest that failure detection should evolve to monitor *multi-dimensional* signals of a system, aka *vital signs* [30, 37, 44]. But, defining signals that represent the health of a system can be tricky. They can be incomplete or too excessive to reason about. Setting accurate thresholds for these signals is also an art. They may be too low to prevent overreacting to benign faults, or too high to reliably detect failures. For example, an impactful service outage in AWS was due to a latent memory leak, which caused the system to get stuck when serving requests and eventually led to a cascading outage [10]. Interestingly, there was a monitor for system memory consumption, but it triggered no alarm because of "the difficulty in setting accurate alarms for a dynamic system" [10]. These monitoring challenges are further aggravated in a multi-tenant environment where both the system and workloads are constantly changing [44].

In this paper, we advocate detecting complex production failures by enhancing *observability* (a measure of how well components' internal states can be inferred from their external interactions [32]). While defining the absolute health or failure of a system in isolation is tricky,

```
void syncWithLeader(long newLeaderZxid) {
  QuorumPacket qp = new QuorumPacket();
  readPacket(qp);
  try {
    if (qp.getType() == Leader.SNAP) {
      deserializeSnapshot(leaderIs);
      String sig = leaderIs.read("signature");
      if (!sig.equals("BenWasHere"))
        throw new IOException("Bad signature");
    } else {
      LOG.error("Unexpected leader packet.");
      System.exit(13);
    }
  } catch (IOException e) {
    LOG.warn("Exception sync with leader", e);
    sock.close();
  }
}
```

Listing 1: A follower requesting a snapshot from the leader tries to *handle* or *log* errors but it does not *report* errors.

modern distributed systems consist of many highly interactive components across layers. So, when a component becomes unhealthy, the issue is likely observable through its effects on the *execution* of some, if not all, other components. For example, in the previous ZooKeeper incident, even though the simple heartbeat detectors did not detect the partial failure, the Cassandra process experienced many request time-outs that caused its own unserved requests to rapidly accumulate. Followers that requested snapshots from the leader also encountered exceptions and could not continue. Thus, errors encountered in the execution path of interactive components enhance the observability of complex failures.

Even though an interactive component (a *requester*) is well-placed to observe issues of another component (a *provider*) when it experiences errors, such a requester is often designed to **handle** that error but not **report** it (e.g., Listing 1). For example, the requester may release a resource, retry a few times, reset its state, use a cached result (i.e., be fail-static), or exit. This tendency to prioritize error handling over error reporting is possibly due to the modularity principle of "separation of concern" [41, 42], which suggests that components should hide as much information as they can and that failure detection and recovery should be each component's own job. Even if a component has incentive to report, it may not have a convenient systematic mechanism to do so. It can write errors in its own logs to be collected and aggregated by a central service, as is done in current practice. The correlation, however, usually happens in an offline troubleshooting phase, which is too late.

We present Panorama, a generic failure detection framework that leverages and enhances system observability to detect complex production failures. It does so by breaking detection boundaries and systematically extracting critical observations from diverse components. Panorama provides unified abstractions and APIs to report observations, and a distributed service to selectively exchange observations. Also, importantly, Panorama keeps the burden on developers low by automatically inserting report-generation code based on offline static analysis. In this way, Panorama automatically converts every component into an observer of the components it interacts with. This construction of *in-situ* observers differentiates Panorama from traditional distributed crash failure detection services [34, 47], which only measure superficial failure indicators.

In applying Panorama to real-world system software, we find some common design patterns that, if not treated appropriately, can reduce observability and lead to misleading observations. For example, if a requester submits requests to a provider, but an indirection layer temporarily buffers the request, the request may appear successful even though the provider has failed. This can cause the requester to report positive evidence about the provider. We study such common design patterns and characterize their impact on system observability (§4). Based on this, we enhance Panorama to recognize these patterns and avoid their effects on observability.

For failure detection, Panorama includes a decision engine to reach a verdict on the status of each component based on reported observations. Because these reports come from errors and successes in the execution paths of requester components instead of artificial, non-service signals, our experience suggests that a simple decision algorithm suffices to reliably detect complex failures.

We have implemented the Panorama system in Go and the static analyzer on top of Soot [46] and AspectJ [2]. Our experiences show that Panorama is easy to integrate with popular distributed systems including ZooKeeper, Cassandra, HDFS, and HBase. Panorama significantly outperforms existing failure detectors in that: (1) it detects crash failures faster; (2) it detects 15 **real-world** gray failures in less than 7 s each, whereas other detectors only detect one in 86 s; (3) Panorama not only detects, but also *locates* failures. Our experiments also show that Panorama is resilient to transient failures and is stable in normal operations. Finally, Panorama introduces only minor overhead (less than 3%) to the systems we evaluate it on.

2 Problem Statement

We consider failure detection in the context of a large distributed system S composed of several subsystems. Each subsystem has multiple components. In total, S contains n processes $P_1, P_2, \ldots, P_n$, each with one or more threads. The whole system lies within a single administrative domain but the code for different system components may be developed by different teams. For example, a stor-

age system may consist of a front-end tier, a distributed lock service, a caching middleware, a messaging service, and a persistence layer. The latter subsystem include metadata servers, structured table servers, and extent data nodes. An extent data node may be multi-threaded, with threads such as a data receiver, a data block scanner, a block pool manager, and an IPC-socket watcher. We assume the components trust each other, collectively providing services to external untrusted applications.

The main goal of failure detection is to correctly report the status of each component; in this work the only components we consider are processes and threads. Traditional failure detectors focus on crash failure, i.e., using only statuses `UP` and `DOWN`. We aim to detect not only crash failure but also gray failure, in which components experience degraded modes "between" `UP` and `DOWN`. The quality of a failure detector is commonly characterized by two properties: *completeness*, which requires that if a component fails, a detector eventually suspects it; and *accuracy*, which requires that a component is not suspected by a detector before it fails. Quality is further characterized by *timeliness*, i.e., how fast true failures are detected. Failure detectors for production systems should also have good *localization*, i.e., ease of pinpointing each failure in a way that enables expedient corrective action.

3 Panorama System

3.1 Overview

At a high level, Panorama takes a collaborative approach: It gathers observations about each component from different sources in real time to detect complex production failures. Collaborative failure detection is not a new idea. Many existing crash-failure detectors such as membership services exchange detection results among multiple components using protocols like gossip [47]. But, the scope of where the detection is done is usually limited to component instances with similar functionality or roles in a particular layer. Panorama pushes the detection scope to an extreme by allowing any thread in any process to report evidence, regardless of its role, layer, or subsystem. The resulting diverse sources of evidence enhance the observability of complex failures.

More importantly, instead of writing separate monitoring code that measures superficial signals, Panorama's philosophy is to leverage *existing code* that lies near the boundaries between different components. Examples of such code include when one thread calls another, and when one process makes an RPC call to another. This captures first-hand observations, especially runtime errors that are generated from the executions of these code regions in production. When Panorama reports a failure, there is concrete evidence and context to help localize where the failure happened.

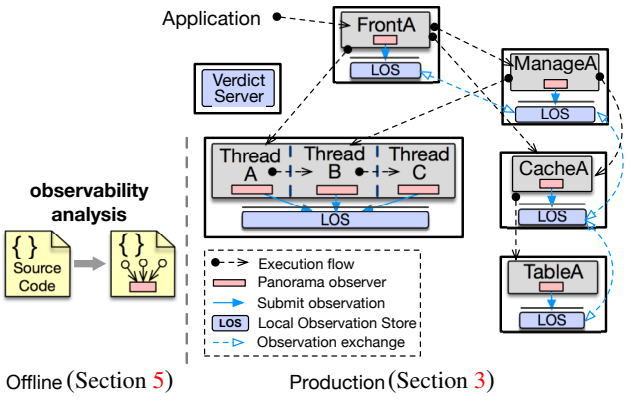

Figure 1: Overview of Panorama. Each Panorama instance runs at the same endpoint with the monitored component.

Figure 1 shows an overview of Panorama. Panorama is a generic detection service that can be plugged into any component in a distributed system. It provides unified abstractions to represent observations about a component's status, and a library for reporting and querying detection results. For scalability, we use a decentralized architecture: for each P_i in a monitored system, a co-located Panorama instance (a separate process) maintains a Local Observation Store (LOS) that stores all the observations that are made either by or about P_i. A local decision engine in the instance analyzes the observations in that LOS and makes a judgment about the process's status. A central verdict server allows easy querying of, and arbitration among, these decentralized LOSes.

The Panorama service depends on many *logical observers* within the running components in the monitored system. Unlike traditional failure detectors, these logical observers are *not* dedicated threads running detection checks. Rather, they are diverse hooks injected into the code. These hooks use a thin library to collect and submit observations to the LOS via local RPC calls. They are inserted offline by a tool that leverages static analysis (§5). To achieve timeliness, the observations are reported in real time as P_i executes. Panorama observers collect evidence not only about the locally attached component, but, more importantly, about other components that the observer interacts with. However, if P_i never interacts with P_j, P_i will not put observations about P_j into its LOS. Panorama runs a dissemination protocol to exchange observations among a clique of LOSes that share common interaction components.

3.2 Abstractions and APIs

To be usable by arbitrary distributed system components, Panorama must provide a unified way to encapsulate ob-

Component	a process or thread
Subject	a component to be monitored
Observer	a component monitoring a subject
Status	the health situation of a subject
Observation	evidence an observer finds of a subject's status
Context	what an observer was doing when it made an observation
Verdict	a decision about a subject's status, obtained by summarizing a set of observations of it

Table 1: Abstractions and terms used in Panorama.

servations for reporting. We now describe our core abstractions and terms, summarized in Table 1.

As discussed earlier, the only components we consider are processes and threads. A component is an *observer* if it makes observations and a *subject* if it is observed; a component may be both an observer and a subject. A *status* is a categorization of the health of a subject; it can be only a small pre-determined set of values, including `HEALTHY`, `DEAD`, and a few levels of `UNHEALTHY`. Another possible value is `PENDING`, the meaning and use of which we will discuss in §5.4.

When an observer sees evidence of a subject's status, that constitutes an *observation*. An observation contains a timestamp of when the observation occurred, the identities of the observer and subject, and the inferred status of the subject. It also contains a *context* describing what the observer was doing when it made the observation, at a sufficient granularity to allow Panorama to achieve fine-grained localization of failures. For instance, the context may include the method the observer was running, or the method's class; the API call the observer was making to the subject; and/or the type of operation, e.g., short-circuit read, snapshot, or row mutation. A *verdict* is a summary, based on a decision algorithm, of a set of observations of the same subject.

Each Panorama instance provides an API based on the above abstractions. It can be invoked by a local component, by another Panorama instance, or by an administration tool. When a component decides to use Panorama, it registers with the local Panorama instance and receives a handle to use for reporting. It reports observations using a local RPC `ReportObservation`; when it is done reporting it unregisters. A Panorama instance can register multiple local observers. If a component does not intend to report observations but merely wants to query component statuses, it need not register.

Each Panorama instance maintains a *watch list*: the set of subjects for which it keeps track of observations. By default, Panorama automatically updates this list to include the components that registered observers interact with. But, each observer can explicitly select subjects for this list using `StartObserving` and `StopObserving`. If another observer in another Panorama instance makes an observation about a subject in the watch list, that observation will be propagated to this instance with a remote RPC `LearnObservation`. Panorama calls `JudgeSubject` each time it collects a new observation, either locally or via remote exchange.

3.3 Local Observation Store

Each Panorama instance maintains a Local Observation Store (LOS) that stores all observation reports made by colocated components. The subjects of these reports include both local and remote components.

The LOS consists of two main structures: the raw observation store and the verdict table. The LOS partitions the raw observation store by subject into multiple tables for efficient concurrent access. Each record in a subject's table corresponds to a single observer; it stores a list of the n most recent observations of that subject made by that observer. The LOS is kept in memory to enable efficient access; asynchronously, its content is persisted to local database to preserve the full observation history, for facilitating troubleshooting later. The raw observation store is synchronized with that of other Panorama instances that share common subjects. Therefore, an LOS contains observations made both locally and remotely.

A local decision engine analyzes the raw observation store to reach a verdict for each subject. This decision result is stored in the verdict table, keyed by subject. The verdict table is *not* synchronized among Panorama instances because it does not have to be: the decision algorithm is deterministic. In other words, given synchronized raw observations, the verdict should be the same. To enable convenient queries over the distributed verdict tables to, e.g., arbitrate among inconsistent verdicts, Panorama uses a central verdict server. Note, though, that the central verdict server is not on any critical path.

Including old observations in decisions can cause misleading verdicts. So, each observation has a Time-to-Live parameter, and a background garbage collection (GC) task runs periodically to retire old observations. Whenever GC changes the observations of a subject, the decision engine re-computes the subject's verdict.

3.4 Observers

Panorama does not employ dedicated failure detectors. Instead, it leverages code logic in existing distributed-system components to turn them into in-situ *logical* observers. Each logical observer's main task is still to provide its original functionality. As it executes, if it encounters an error related to another component, in addition to handling the error it will also report it as an observation to Panorama. There are two approaches to turn

a component into a Panorama observer. One is to insert Panorama API hooks into the component's source code. Another is to integrate with the component's logs by continuously parsing and monitoring log entries related to other components. The latter approach is transparent to components but captures less accurate information. We initially adopted the latter approach by adding plug-in support in Panorama to manage log-parsing scripts. But, as we applied Panorama to more systems, maintaining these scripts became painful because their logging practices differed significantly. Much information is also unavailable in logs [50]. Thus, even though we still support logging integration, we mainly use the instrumentation approach. To relieve developers of the burden of inserting Panorama hooks, Panorama provides an offline analysis tool that does the source-code instrumentation automatically. §4 describes this offline analysis.

3.5 Observation Exchange

Observations submitted to the LOS by a local observer only reflect a partial view of the subject. To reduce bias in observations, Panorama runs a dissemination protocol to propagate observations to, and learn observations from, other LOSes. Consequently, for each monitored subject, the LOS stores observations from multiple observers. The observation exchange in Panorama is only among cliques of LOSes that share a subject. To achieve selective exchange, each LOS keeps a *watch list*, which initially contains only the local observer. When a local observer reports an observation to the LOS, the LOS will add the observation's subject to the watch list to indicate that it is now interested in others' observations about this subject. Each LOS also keeps an *ignore list* for each subject, which lists LOSes to which it should not propagate new observations about that subject. When a local observation for a new subject appears for the first time, the LOS does a one-time broadcast. LOSes that are not interested in the observation (based on their own watch lists) will instruct the broadcasting LOS to include them in its ignore list. If an LOS later becomes interested in this subject, the protocol ensures that the clique members remove this LOS from their ignore lists.

3.6 Judging Failure from Observations

With numerous observations collected about a subject, Panorama uses a decision engine to reach a verdict and stores the result in the LOS's verdict table. A simple decision policy is to use the latest observation as the verdict. But, this can be problematic since a subject experiencing intermittent errors may be treated as healthy. An alternative is to reach an unhealthy verdict if there is *any* recent negative observation. This could cause one biased observer, whose negative observation is due to its own issue, to mislead others.

We use a bounded-look-back majority algorithm, as follows. For a set of observations about a *subject*, we first group the observations by the unique *observer*, and analyze each group separately. The observations in a group are inspected from latest to earliest and aggregated based on their associated *contexts*. For an observation being inspected, if its *status* is different than the previously recorded status for that context, the look-back of observations for that context stops after a few steps to favor newer statuses. Afterwards, for each recorded context, if either the latest status is unhealthy or the healthy status does not have the strict majority, the verdict for that context is unhealthy with an aggregated severity level.

In this way, we obtain an analysis summary for each context in each group. To reach a final verdict for each context across all groups, the summaries from different observers are aggregated and decided based on a simple majority. Using group-based summaries allows incremental update of the verdict and avoids being biased by one observer or context in the aggregation. The decision engine could use more complex algorithms, but we find that our simple algorithm works well in practice. This is because most observations collected by Panorama constitute strong evidence rather than superficial signals.

The `PENDING` status (Section 4.3) needs additional handling: during the look-back for a context, if the current status is `HEALTHY` and the older status is `PENDING`, that older `PENDING` status will be skipped because it was only temporary. In other words, that partial observation is now complete. Afterwards, a `PENDING` status with occurrences exceeding a threshold is downgraded to `UNHEALTHY`.

4 Design Pattern and Observability

The effectiveness of Panorama depends on the hooks in observers. We initially designed a straightforward method to insert these hooks. In testing it on real-world distributed systems, however, we found that component interactions in practice can be complex. Certain interactions, if not treated appropriately, will cause the extracted observations to be misleading. In this section, we first show a gray failure that our original method failed to detect, and then investigate the reason behind the challenge.

4.1 A Failed Case

In one incident of a production ZooKeeper service, applications were experiencing many lock timeouts [23]. An engineer investigated the issue by checking metrics in the monitoring system and found that the number of connections per client had significantly increased. It ini-

tially looked like a resource leak in the client library, but the root cause turned out to be complicated.

The production environment used IPSec to secure inter-host traffic, and a Linux kernel module used Intel AES instructions to provide AES encryption for IPSec. But this kernel module could occasionally introduce data corruption with Xen paravirtualization, for reasons still not known today. Typically the kernel validated packet checksums and dropped corrupt packets. But, in IPSec, two checksums exist: one for the IP payload, the other for the encrypted TCP payload. For IPSec NAT-T mode, the Linux kernel did not validate the TCP payload checksum, thereby permitting corrupt packets. These were delivered to the ZooKeeper leader, including a corrupted length field for a string. When ZooKeeper used the length to allocate memory to deserialize the string, it raised an out-of-memory (OOM) exception.

Surprisingly, when this OOM exception happened, ZooKeeper continued to run. Heartbeats were normal and no leader re-election was triggered. When evaluating this incident in Panorama, no failure was reported either. We studied the ZooKeeper source code to understand why this happened. In ZooKeeper, a request is first picked up by the listener thread, which then calls the `ZooKeeperServer` thread that further invokes a chain of `XXXRequestProcessor` threads to process the request. The OOM exception happens in the `PrepRequestProcessor` thread, the first request processor. The `ZooKeeperServer` thread invokes the interface of the `PrepRequestProcessor` as follows:

```
1  try {
2    firstProcessor.processRequest(si);
3  } catch (RequestProcessorException e) {
4    LOG.error("Unable to process request: " + e);
5  }
```

If the execution passes line 2, it provides positive evidence that the `PrepRequestProcessor` thread is healthy. If, instead, the execution reaches line 4, it represents negative evidence about `PrepRequestProcessor`. But with the Panorama hooks inserted at both places, no negative observations are reported. This is because the implementation of the `processRequest` API involves an indirection: it simply puts a request in a queue and immediately returns. Asynchronously, the thread polls and processes the queue. Because of this design, even though the OOM exception causes the `PrepRequestProcessor` thread to exit its main loop, the `ZooKeeperServer` thread is still able to call `processRequest` and is unable to tell that `PrepRequestProcessor` has an issue. The hooks are only observing the status of the indirection layer, i.e., the queue, rather than the `PrepRequestProcessor` thread. Thus, negative observations only appear when the request queue cannot insert new items; but, by default, its capacity is `Integer.MAX_VALUE`!

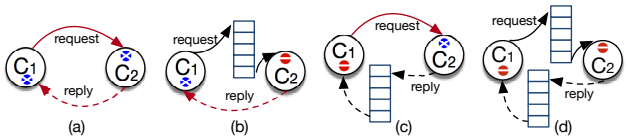

Figure 2: Design patterns of component interactions and their impact on failure observability. ⛒ means that failure is observable to the other component, and ⛔ means that failure is unobservable to it.

4.2 Observability Patterns

Although the above case is a unique incident, we extrapolate a deeper implication for failure detection: certain design patterns can undermine failure observability in a system and thereby pose challenges for failure detection. To reveal this connection, consider two components C_1 and C_2 where C_1 makes requests of C_2. We expect that, through this interaction, C_1 and C_2 should be able to make observations about each other's status. However, their style of interaction can have a significant effect on this observability.

We have identified the following four basic patterns of interaction (Figure 2), each having a different effect on this observability. Interestingly, we find examples of all four patterns in real-world system software.

(a) No indirection. Pattern (a) is the most straightforward. C_1 makes a request to C_2, then C_2 optionally replies to C_1. This pattern has the best degree of observability: C_1 can observe C_2 from errors in its request path; C_2 can also observe C_1 to some extent in its reply path. Listing 1 shows an example of this pattern. In this case, C_1 is the follower and C_2 is the leader. C_1 first contacts C_2, then C_2 sends C_1 a snapshot or other information through an input stream. Failures are observed via errors or timeouts in the connection, I/O through the input stream, and/or reply contents.

(b) Request indirection. A level of indirection exists in the request path: when C_1 makes a request to C_2, an intermediate layer (e.g., a proxy or a queue) takes the request and replies to C_1. C_2 will later take the request from the intermediate layer, process it, and optionally reply to C_1 directly. This design pattern has a performance benefit for both C_1 and C_2. It also provides decoupling between their two threads. But, because of the indirection, C_1 no longer directly interacts with C_2 so C_2's observability is reduced. The immediate observation C_1 makes when requesting from C_2 does not reveal whether C_2 is having problems, since usually the request path succeeds as in the case in §4.1.

(c) Reply indirection. Pattern (c) is not intuitive. C_1 makes a request, which is directly handled by C_2, but the reply goes through a layer of indirection (e.g., a queue or a proxy). Thus, C_1 can observe issues in C_2 but C_1's ob-

servability to C_2 is reduced. One scenario leading to this pattern is when a component makes requests to multiple components and needs to collect more than one of their replies to proceed. In this case, replies are queued so that they can be processed en masse when a sufficient number are available. For example, in Cassandra, when a process sends digest requests to multiple replicas, it must wait for responses from R replicas. So, whenever it gets a reply from a replica, it queues the reply for later processing.

(d) Full indirection. In pattern (d), neither component directly interacts with the other so they get the least observability. This pattern has a performance benefit since all operations are asynchronous. But, the code logic can be complex. ZooKeeper contains an example: When a follower forwards a request to a leader, the request is processed asynchronously, and when the leader later notifies the follower to commit the request, that notification gets queued.

4.3 Implications

Pattern (a) has the best failure observability and is easiest for Panorama to leverage. The other three patterns are more challenging; placing observation hooks without considering the effects of indirection can cause incompleteness (though not inaccuracy) in failure detection (§2). That is, a positive observation will not necessarily mean the monitored component is healthy but a negative observation means the component is unhealthy. Pragmatically, this would be an acceptable limitation if the three indirection patterns were uncommon. However, we checked the cross-thread interaction code in several distributed systems and found, empirically, that patterns (a) and (b) are both pervasive. We also found that different software has different preferences, e.g., ZooKeeper uses pattern (a) frequently, but Cassandra uses pattern (b) more often.

This suggests Panorama should accommodate indirection in extracting observations. One solution is to instrument hooks in the indirection layer. But, we find that indirection layers in practice are implemented with various data structures and are often used for multiple purposes, making tracking difficult. We use a simple but robust solution and describe it in §5.4.

5 Observability Analysis

To systematically identify and extract useful observations from a component, Panorama provides an offline tool that statically analyzes a program's source code, finds critical points, and injects hooks for reporting observations.

5.1 Locate Observation Boundary

Runtime errors are useful evidence of failure. Even if an error is tolerated by a requester, it may still indicate a critical issue in the provider. But, not all errors should be reported. Panorama only extracts errors generated when crossing component boundaries, because these constitute observations from the requester side. We call such domain-crossing function invocations *observation boundaries*.

The first step of observability analysis is to locate observation boundaries. There are two types of such boundaries: inter-process and inter-thread. An inter-process boundary typically manifests as a library API invocation, a socket I/O call, or a remote procedure call (RPC). Sometimes, it involves calling into custom code that encapsulates one of those three to provide a higher-level messaging service. In any case, with some domain knowledge about the communication mechanisms used, the analyzer can locate inter-process observation boundaries in source code. An inter-thread boundary is a call crossing two threads within a process. The analyzer identifies such boundaries by finding custom public methods in classes that extend the thread class.

5.2 Identify Observer and Observed

At each observation boundary, we must identify the observer and subject. Both identities are specific to the distributed system being monitored. For thread-level observation boundaries, the thread identities are statically analyzable, e.g., the name of the thread or class that provides the public interfaces. For process-level boundaries, the observer identity is the process's own identity in the distributed system, which is known when the process starts; it only requires one-time registration with Panorama. We can also usually identify the subject identity, if the remote invocations use well-known methods, via either an argument of the function invocation or a field in the class. A challenge is that sometimes, due to nested polymorphism, the subject identity may be located deep down in the type hierarchy. For example, it is not easy to determine if `OutputStream.write()` performs network I/O or local disk I/O. We address this challenge by changing the constructors of remote types (e.g., socket get I/O stream) to return a compatible wrapper that extends the return type with a subject field and can be differentiated from other types at runtime by checking if that field is set.

5.3 Extract Observation

Once we have observation boundaries, the next step is to search near them for *observation points*: program points that can supply critical evidence about observed components. A typical example of such an observation point is

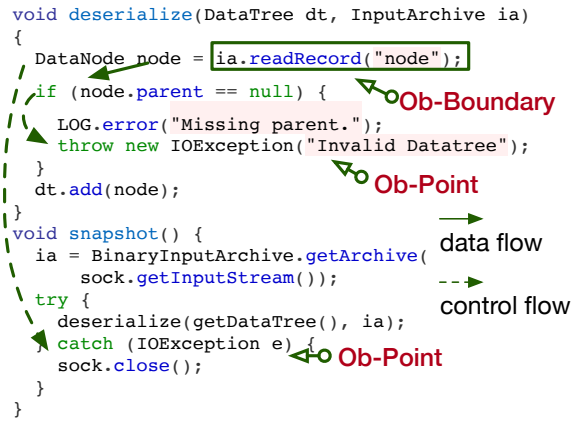

Figure 3: Observation points in direct interaction (§4.2).

an exception handler invoked when an exception occurs at an observation boundary.

To locate observation points that are exception handlers, a straightforward approach is to first identify the type of exceptions an observation boundary can generate, then locate the `catch` clauses for these types in code regions after the boundary. There are two challenges with this approach. First, as shown in Figure 3, an exception could be caught at the caller or caller's caller. Recursively walking up the call chain to locate the clause is cumbersome and could be inaccurate. Second, the type of exception thrown by the boundary could be a generic exception such as `IOException` that could be generated by other non-boundary code in the same `try` clause. These two challenges can be addressed by inserting a `try` just before the boundary and a `catch` right after it. This works but, if the observation boundaries are frequent, the excessive wrapping can cause non-trivial overhead.

The ideal place to instrument is the shared exception handler for adjacent invocations. Our solution is to add a special field in the base `Throwable` class to indicate the subject identity and the context, and to ensure boundary-generated exceptions set this field. Then, when an exception handler is triggered at runtime, we can check if this field is set, and if so treat it as an observation point. We achieve the field setting by wrapping the outermost function body of each boundary method with a `try` and `catch`, and by rethrowing the exception after the hook. Note that this preserves the original program semantics.

Another type of observation point we look for is one where the program handles a response received from across a boundary. For example, the program may raise an exception for a missing field or wrong signature in the returned `DataNode` in Figure 3, indicating potential partial failure or corrupt state in the remote process. To locate these observation points, our analyzer performs intra-procedural analysis to follow the data flow of responses from a boundary. If an exception thrown is control-dependent on the response, we consider it an observation point, and we insert code to set the subject/context field before throwing the exception just as described earlier. This data-flow analysis is conservative: e.g., the code `if (a + b > 100) {throw Exception("unexpected");}`, where `a` comes from a boundary but `b` does not, is not considered an observation point because the exception could be due to `b`. In other words, our analysis may miss some observation points but will not locate wrong observation points.

So far, we have described negative observation points, but we also need mechanisms to make positive observations. Ideally, each successful interaction across a boundary is an observation point that can report positive evidence. But, if these boundaries appear frequently, the positive observation points can be excessive. So, we coalesce similar positive observation points that are located close together.

For each observation point, the analyzer inserts hooks to discover evidence and report it. At each negative observation point, we get the subject identity and context from the modified exception instance. We statically choose the status; if the status is to be some level of `UNHEALTHY` then we set this level based on the severity of the exception handling. For example, if the exception handler calls `System.exit()`, we set the status to a high level of `UNHEALTHY`. At each positive observation point, we get the context from the nearby boundary and also statically choose the status. We immediately report each observation to the Panorama library, but the library will typically not report it synchronously. The library will buffer excessive observations and send them in one aggregate message later.

5.4 Handling Indirection

As we discussed in §4, observability can be reduced when indirection exists at an observation boundary. For instance, extracted observations may report the subject as healthy while it is in fact unhealthy. The core issue is that indirection *splits* a single interaction between components among multiple observation boundaries. A successful result at the first observation boundary may only indicate partial success of the overall interaction; the interaction may only truly complete later, when, e.g., a callback is invoked, or a condition variable unblocks, or a timeout occurs. We must ideally wait for an interaction to complete before making an observation.

We call the two locations of a split interaction the *ob-origin* and *ob-sink*, reflecting the order they're encountered. Observations at the ob-origin represent positive but temporary and weak evidence. For example, in Figure 4, the return from `sendRR` is an ob-origin. Where the

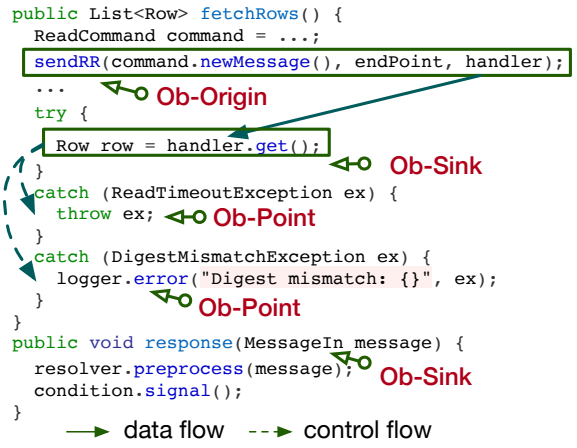

Figure 4: Observation points when indirection exists (§4.2).

callback of `handler`, `response`, is invoked, it is an ob-sink. In addition, when the program later blocks waiting for the callback, e.g., `handler.get`, the successful return is also an ob-sink. If an ob-origin is properly matched with an ob-sink, the positive observation becomes complete and strong. Otherwise, an outstanding ob-origin is only a weak observation and may degrade to a negative observation, e.g., when `handler.get` times out.

Tracking an interaction split across multiple program locations is challenging given the variety of indirection implementations. To properly place hooks when indirection exists, the Panorama analyzer needs to know what methods are asynchronous and the mechanisms for notification. For instance, a commonly used one is Java FutureTask [40]. For custom methods, this knowledge comes from specifications of the boundary-crossing interfaces, which only requires moderate annotation. With this knowledge, the analyzer considers an ob-origin to be immediately after any call site of an asynchronous interface. We next discuss how to locate ob-sinks.

We surveyed the source code of popular distributed systems and found the majority of ob-sinks fall into four patterns: (1) invoking a callback-setting method; (2) performing a blocking wait on a callback method; (3) checking a completion flag; and (4) reaching another observation boundary with a third component, in cases when a request must be passed on further. For the first two patterns, the analyzer considers the ob-sink to be before and after the method invocation, respectively. For the third pattern, the analyzer locates the spin-loop body and considers the ob-sink to be immediately after the loop. The last pattern resembles SEDA [48]: after A asynchronously sends a request to B, B does not notify A of the status after it finishes but rather passes on the request to C. Therefore, for that observation boundary in B, the analyzer needs to not only insert a hook for C but also treat it as an ob-sink for the A-to-B interaction.

When our analyzer finds an ob-origin, it inserts a hook that submits an observation with the special status `PENDING`. This means that the observer currently only sees weak positive evidence about the subject's status, but expects to receive stronger evidence shortly. At any ob-sink indicating positive evidence, our analyzer inserts a hook to report a `HEALTHY` observation. At any ob-sink indicating negative evidence, the analyzer inserts a hook to report a negative observation.

To link an ob-sink observation with its corresponding ob-origin observation, these observations must share the same subject and context. To ensure this, the analyzer uses a similar technique as in exception tracking. It adds a special field containing the subject identity and context to the callback handler, and inserts code to set this field at the ob-origin. If the callback is not instrumentable, e.g., because it is an integer resource handle, then the analyzer inserts a call to the Panorama library to associate the handle with an identity and context.

Sometimes, the analyzer finds an ob-origin but cannot find the corresponding ob-sink or cannot extract the subject identity or context. This can happen due to either lack of knowledge or the developers having forgotten to check for completion in the code. In such a case, the analyzer will not instrument the ob-origin, to avoid making misleading `PENDING` observations.

We find that ob-origin and ob-sink separation is useful in detecting not only issues involving indirection but also liveness issues. To see why, consider what happens when A invokes a boundary-crossing blocking function of B, and B gets stuck so the function never returns. When this happens, even though A witnesses B's problem, it does not get a chance to report the issue because it never reaches the observation point following the blocking call. Inserting an ob-origin before the function call provides evidence of the liveness issue: LOSes will see an old `PENDING` observation with no subsequent corresponding ob-sink observation. Thus, besides asynchronous interfaces, call sites of synchronous interfaces that may block for long should also be included in the ob-origin set.

6 Implementation

We implemented the Panorama service in $\sim 6,000$ lines of Go code, and implemented the observability analyzer (§5) using the Soot analysis framework [46] and the AspectJ instrumentation framework [2].

We defined Panorama's interfaces using protocol buffers [7]. We then used the gRPC framework [5] to build the RPC service and to generate clients in different languages. So, the system can be easily used by various components written in different languages. Panorama provides a thin library that wraps the gRPC client for

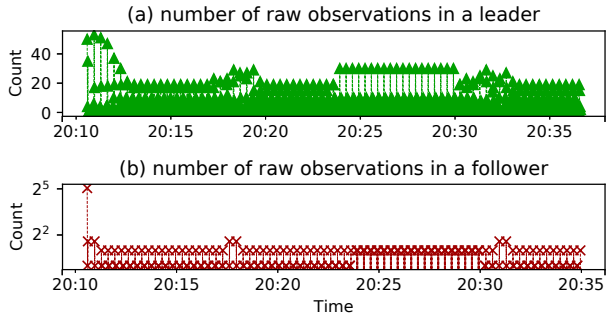

Figure 5: Number of raw observations in two Panorama observers. Each data point represents one second.

efficient observation reporting; each process participating in observation reporting is linked with this library. The thin library provides features such as asynchronous reporting, buffering and aggregation of frequent observations, identity resolution, rate limiting, quick cancellation of `PENDING` statuses, and mapping of ob-sink handles (§5.4). So, most operations related to observation reporting do not directly trigger local RPC calls to Panorama; this keeps performance impact low.

7 Evaluation

In this section, we evaluate our Panorama prototype to answer several key questions: (1) Can observations be systematically captured? (2) Can observation capturing detect regular failures? (3) Can Panorama detect production gray failures? (4) How do transient failures affect Panorama? (5) How much overhead does an observer incur by participating in the Panorama service?

7.1 Experiment Setup

We run our experiments in a cluster of 20 physical nodes. Each machine has a 2.4 GHz 10-core Intel Xeon E5-2640v4 CPU, 64 GB of RAM, and a 480 GB SATA SSD; they all connect to a single 10 Gbps Ethernet switch. They run Ubuntu 16.04 with Linux kernel version 4.4.0. We evaluate Panorama with four widely-used distributed systems: ZooKeeper, Hadoop, HBase, and Cassandra. HBase uses HDFS for storing data and ZooKeeper for coordination, so an HBase setup resembles a service with multiple subsystems. We continuously exercise these services with various benchmark workloads to represent an active production environment.

7.2 Integration with Several Systems

Panorama provides a generic observation and failure detection service. To evaluate its generality, we apply it to ZooKeeper, HDFS, Hadoop, HBase, and Cassandra, at both process and thread level. The integration is successful without significant effort or changes to the system design. Our simple abstractions and APIs (§3.2) naturally support various types of failure evidence in each system. For instance, we support semantic errors, such as responses with missing signatures; generic errors, such as remote I/O exceptions; and liveness issues, such as indefinite blocking or custom time-outs. The integration is enabled by the observability analyzer (§5). In applying the analyzer to a system, we need annotations about what boundary-crossing methods to start with, what methods involve indirection, and what patterns it uses (§5.4). The annotation effort to support this is moderate (Table 2). HDFS requires the most annotation effort, which took one author about 1.5 days to understand the HDFS source code, identify the interfaces and write annotation specification. Fortunately, most of these boundary-crossing methods remain stable over releases. When running the observability analysis, Cassandra is more challenging to analyze compared to the others since it frequently uses indirection. On the other hand, its mechanisms are also well-organized, which makes the analysis systematic. The observability analysis is mainly intra-procedural and can finish instrumentation within 10 seconds for each of the four systems (Table 2). Figure 5 shows the observations collected from two instrumented processes in ZooKeeper. The figure also shows that the observations made change as the observer executes, and depend on the process's interaction patterns.

	ZooKeeper	Cassandra	HDFS	HBase
Annotations	24	34	65	16
Analysis Time	4.2	6.8	9.9	7.5

Table 2: Annotations and analysis time (in seconds).

7.3 Detection of Crash Failures

Panorama aims to detect complex failures not limited to fail-stop. As a sanity check on the effectiveness of its detection capability, we first evaluate how well Panorama detects fail-stop failures. To measure this, we inject various fail-stop faults including process crashes, node shutdowns, and network disconnections. Table 3 shows the detection time for ten representative crash-failure cases: failures injected into the ZooKeeper leader, ZooKeeper follower, Cassandra data node, Cassandra seed node, HDFS name node, HDFS data node, HBase master and HBase regionserver. We see that with Panorama the observers take less than 10 s to detect all ten cases, and indeed take less than 10 ms to detect all ZooKeeper failures. The observers make the observations leading to these detections when, while interacting with the

Detector	Crash Failure Injection Site							
	ZooKeeper		Cassandra		HDFS		HBase	
	leader	follower	seed	datanode	namenode	datanode	master	regionserver
Built-in	13 ms	3 ms	28 s	26 s	708 ms	30 s (12 min*)	11 s	102 ms
Panorama	8 ms	2 ms	8 s	9 s	723 ms	6 s	1.5 s	102 ms

Table 3: Crash-failure detection time. **The name node marks the data node stale in* 30 s*, and dead in* 12 min.

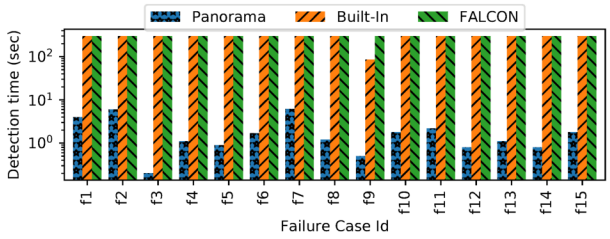

Figure 6: Detection time for gray failures in Table 4.

ID	System	Fault Synopsis
f1	ZooKeeper	faulty disk in leader causes cluster lock-up
f2	ZooKeeper	transient network partition leads to prolonged failures in serving requests
f3	ZooKeeper	corrupted packet in de-serialization
f4	ZooKeeper	transaction thread exception
f5	ZooKeeper	leader fails to write transaction log
f6	Cassandra	response drop blocks repair operations
f7	Cassandra	stale data in leads to wrong node states
f8	Cassandra	streaming silently fail on unexpected error
f9	Cassandra	commitlog executor exit causes GC storm
f10	HDFS	thread pool exhaustion in master
f11	HDFS	failed pipeline creation prevents recovery
f12	HDFS	short circuit reads blocked due to death of domain socket watcher
f13	HDFS	blockpool fails to initialize but continues
f14	HBase	dead root drive on region server
f15	HBase	replication stalls with empty WAL files

Table 4: Evaluated *real-world* gray failures. In all cases, some severe service disruption occured (e.g., all create requests failed) while the failing component was perceived to be healthy.

failed components, they experience either request/response time-outs or I/O exceptions.

As a basis for comparison, we also measure failure detection time when using the failure detectors built into these systems. We find that for ZooKeeper, Panorama detects the failures slightly faster than the built-in detector, while for Cassandra, HDFS datanode and HBase master, Panorama achieves much faster detection time. This is because, to tolerate asynchrony, Cassandra and HDFS use conservative settings for declaring failures based on loss of heartbeats. For HDFS namenode, we use a High-Availability setup that leverages ZooKeeper for failure detection (when a ZooKeeper ephemeral node expires). Under this setup, the built-in detector achieves a slightly faster time than Panorama because the ZooKeeper service is co-located with HDFS, whereas Panorama's detection is from observations made by remote datanodes.

7.4 Detection of Gray Failures

To evaluate Panorama's ability to detect complex failures, we reproduce 15 **real-world** production gray failures from ZooKeeper, HDFS, HBase, and Cassandra, described in Table 4. Each of these caused severe service disruption, e.g., all write requests would fail. Worse still, in each case the system was perceived as healthy, so no recovery actions were taken during the resulting outage.

Panorama is able to detect the gray failure for **all** 15 cases. Figure 6 shows Panorama's detection time (in seconds) for each case. We often find that a failure is observed and reported by multiple observers; we use the first failure observation's timestamp in a final verdict as the detection time. The detection times have a minimum of 0.2 s and a maximum of 7 s, with the majority smaller than 3 s. The intra-process observers tend to capture failure evidence faster than the inter-process observers. For all cases, the failure evidence clearly stands out in the observations collected about the sick process, so the decision algorithm (§3.6) requires no special tuning.

We compare Panorama with three baselines: the system's built-in failure detector, Falcon [34], and the φ accrual detector [29]. As shown in Figure 6, in all but one case, no baseline detects the gray failure within 300 s. That one case is f9, where Cassandra's built-in detector, a form of the φ detector with some application state, reports failure after 86 s when the partial fault of the Cassandra commitlog executor component eventually degrades into a complete failure due to uncommitted writes piling up on the JVM heap and causing the process to spend most of its time doing garbage collection.

Figure 7 shows a detailed timeline of the detection of gray failure f1. We see that the observers (in this case the followers) quickly gather failure evidence while interacting with the unhealthy leader. Also, when the leader's fault is gone, those observers quickly gather positive evi-

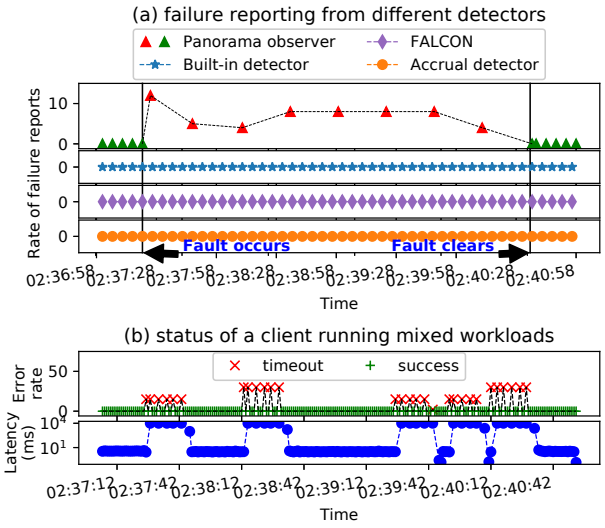

Figure 7: Timeline in detecting gray failure f1 from Table 4.

Figure 8: Verdict during transient failures.

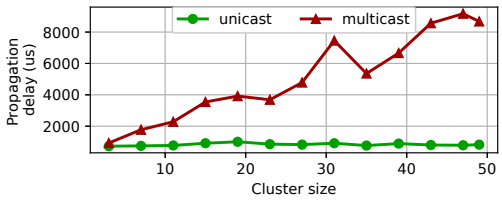

Figure 9: Scalability of observation propagation latency. "unicast": propagate an observation to a single Panorama instance; "multicast": propagate an observation to all interested Panorama instances.

dence that clears the failure observation. During the failure period, no other baseline reports failure. Figure 7 also shows the view from a ZooKeeper client that we run continuously throughout the experiment as a reference. We can see Panorama's reporting closely matches the experience of this client. Interestingly, since the gray failure mainly impacts write requests but the client executes a mixture of read and write requests, its view is not very stable; nevertheless, Panorama consistently reports a verdict of `UNHEALTHY` during the failure period.

7.5 Fault Localization

In addition to detecting the 15 production failures quickly, Panorama also pinpoints each failure with detailed context and observer (§3.2) information. This localization capability allows administrators to interpret the detection results with confidence and take concrete actions. For example, in detecting the crash failure in the ZooKeeper follower, the verdict for the leader is based on observations such as `|peer@3,peer@5,peer@8| 2018-03-23T02:28:58.873 {Learner: U,RecvWorker: U,QuorumCnxManager: U}`, which identify the observer as well as the contexts `Learner`, `RecvWorker`, and `QuorumCnxManager`. In detecting gray failure f1, the negative observations of the unhealthy leader are associated with three contexts `SerializeUtils`, `DataTree`, and `StatPersisted`; this localizes the failure to the serialization thread in leader.

7.6 Transient Failure, Normal Operations

Because Panorama can gather observations from any component in a system, there is a potential concern that noisy observations will lead to many false alarms. But, empirically, we find that this does not happen. The Panorama analyzer assigns the context of an observation properly to avoid falsely aggregating observations made in interacting with different functionalities of a complex process. The simple decision algorithm in §3.6 is robust enough to prevent a few biased observers or transient failures from dominating the verdict. Figure 8 shows the verdict for the ZooKeeper leader in an experiment. A few followers report transient faults about the leader in one context, so Panorama decides on a negative verdict. But, within a few seconds, the verdict changes due to positive observations and expiration of negative observations. Panorama then judges the leader as healthy for the remainder of the experiment, which matches the truth.

We deploy Panorama with ZooKeeper and run for 25 hours, during which multiple ZooKeeper clients continuously run various workloads non-stop to emulate normal operations in a production environment. In total, Panorama generates 797,219 verdicts, with all but 705 (0.08%) of them being `HEALTHY`; this is a low false alarm rate. In fact, all of the negative observations are made in the first 22 seconds, during which the system is bootstrapping and unstable. After the 22 seconds, no negative observations are reported for the remaining 25 hours.

We also inject minor faults including overloaded component, load spike and transient network partition that are modeled after two production ZooKeeper and HDFS traces. These minor faults do not affect the regular service. We find Panorama overall is resilient to these noises in reaching a verdict. For example, an overloaded ZooKeeper follower made a series of misleading obser-

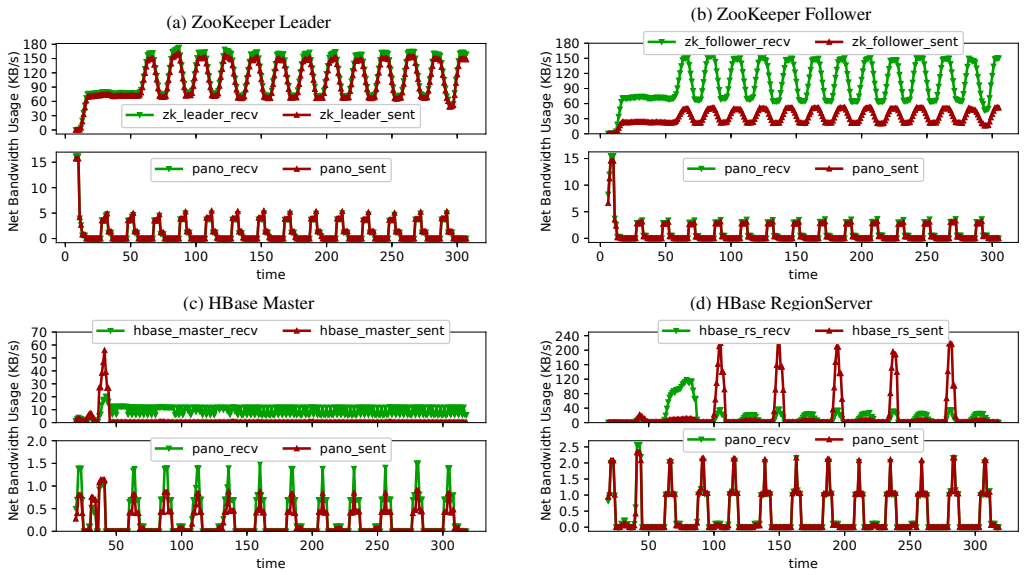

Figure 10: Network bandwidth usage of the Panorama instance and its monitored component.

Report	ReportAsync	Judge	Propagate
114.6 µs	0.36 µs	109.0 µs	776.3 µs

Table 5: Average latency of major operations in Panorama.

vations that the leader is `UNHEALTHY`. But these biased observations from a single observer did not result in a verdict of `UNHEALTHY` status for the leader. When there were many such overloaded followers, however, the leader was falsely convicted as `UNHEALTHY` even though the actual issues were within the observers.

7.7 Performance

Table 5 shows microbenchmark results: how long four major operations in Panorama take on average. Reporting an observation to Panorama only requires a local RPC, so the average latency of reporting is fast (around 100 µs). And, the asynchronous API for reporting takes even less time: on average less than 1 µs. Propagation of an observation to another Panorama instance takes around 800 µs. Figure 9 shows how the propagation latency changes as the cluster size increases.

When a Panorama instance is active, the CPU utilization attributable to it is on average 0.7%. For each monitored subject, the number of observations kept in LOS is bounded so the memory usage is close to a constant. Thus, the total memory usage depends on the number of monitored subjects. When we measure the ZooKeeper deployment with Panorama, and find that the heap memory allocation stabilizes at ∼7 MB for a moderately active instance, and at ∼46 MB for a highly active instance. The network bandwidth usage of Panorama instance for exchanging observations is small compared to the bandwidth usage of the monitored components (Figure 10).

System	Latency		Throughput	
	Read	Write	Read	Write
ZK	69.5 µs	1435 µs	14 402 op/s	697 op/s
ZK+	70.6 µs	1475 µs	14 181 op/s	678 op/s
C*	677 µs	680 µs	812 op/s	810 op/s
C*+	695 µs	689 µs	802 op/s	804 op/s
HDFS	51.0 s	61.0 s	423 MB/s	88 MB/s
HDFS+	52.5 s	62.2 s	415 MB/s	86 MB/s
HBase	746 µs	1682 µs	1172 op/s	549 op/s
HBase+	748 µs	1699 µs	1167 op/s	542 op/s

Table 6: Performance of the original system versus the performance of the system instrumented with Panorama hooks (System+). ZK stands for ZooKeeper and C* stands for Cassandra. The latency results for HDFS are total execution times.

We test the end-to-end request latency and throughput impact of integrating with Panorama for HDFS, ZooKeeper, HBase, and Cassandra, using YCSB [16], DFSIO and a custom benchmark tool. Table 6 shows the results. The latency increase and throughout decrease for each system is below 3%. We achieve this low overhead because the reporting API is fast and because most hooks are in error-handling code, which is not triggered in normal operation. The positive-observation hooks lie in the normal execution path, but their cost is reduced by coalescing the hooks with the analyzer (§5.3) and batching the reporting with the thin client library. Without this optimization, the performance overhead can be up to 18%.

8 Discussion and Limitations

Panorama proposes a new way of building failure detection service by constructing *in-situ* observers. The evaluation results demonstrate the effectiveness of leveraging observability for detecting complex production failures. The process of integrating Panorama with real-world distributed systems also makes us realize how the diverse programming paradigms affect systems observability. For example, HDFS has a method `createBlockOutputStream` that takes a list of data nodes as argument and creates a pipeline among them; if this method fails, it indicates one of the data nodes in the pipeline is problematic. From observability point of view, if a negative evidence is observed through this method, it is associated with multiple possible subjects. Fortunately, an `errorIndex` variable is maintained internally to indicate which data node causes the error, which can be used to determine the exact subject. It is valuable to investigate how to modularize a system and design its interfaces to make it easier to capture failure observability.

There are several limitations of Panorama that we plan to address in future work. First, Panorama currently focuses on failure detection. To improve end-to-end availability, we plan to integrate the detection results with failure recovery actions. Second, Panorama currently does not track causality. Enhancing observations with causality information will be useful for correctly detecting and pinpointing failing components in large-scale cascading failures. Third, we plan to add support for languages other than Java to the Panorama analyzer, and evaluate it with a broader set of distributed systems.

9 Related Work

Failure Detection. There is an extensive body of work on studying and improving failure detection for distributed systems [8, 13, 14, 20, 29, 47]. A recent prominent work in this space is Falcon [34], in which the authors argue that a perfect failure detector (PFD) can be built [9] by replacing end-to-end timeouts with layers of spies that can kill slow processes. Panorama is complimentary to these efforts, which mainly focus on detecting crash failures. Panorama's goal is to detect complex production failures [11, 25, 30]. In terms of approach, Panorama is unique in enhancing system observability by constructing *in-situ* observers in place of any component's code, instead of using dedicated detectors such as spies or sensors that are outside components' normal execution paths.

Monitoring and Tracing. Improving monitoring and tracing of production systems is also an active research area. Examples include Magpie [12], X-Trace [21], Dapper [45] and Pivot Tracing [35]. The pervasive metrics collected by these systems enhance system observability, and their powerful tracing capabilities may help Panorama better deal with the indirection challenge (§4). But they are massive and difficult to reason about [15, 37, 44]. Panorama, in contrast, leverages errors and exceptions generated from an observer's normal execution to report complex but serious failures.

Accountability. Accountability is useful for detecting Byzantine component behavior in a distributed system [28, 51]. PeerReview [27] provides accountability by having other nodes collecting evidence about the correctness of a node through their message exchanges. Panorama's approach is inspired by PeerReview in that it also leverages evidence about other components in a system. But Panorama mainly targets production gray failures instead of Byzantine faults. Unlike PeerReview, Panorama places observability hooks in the existing code of a component and does not require a reference implementation or a special protocol.

10 Conclusion

We present Panorama, a system for detecting production failures in distributed systems. The key insight enabling Panorama is that system observability can be enhanced by automatically turning each component into an observer of the other components with which it interacts. By leveraging these first-hand observations, a simple detection algorithm can achieve high detection accuracy. In building Panorama, we further discover observability patterns and address the challenge of reduced observability due to indirection. We implement Panorama and evaluate it, showing that it introduces minimal overhead to existing systems. Panorama can detect and localize 15 real-world gray failures in less than 7 s, whereas existing detectors only detect one of them in under 300 s. The source code of Panorama system is available at `https://github.com/ryanphuang/panorama`.

Acknowledgments

We thank the OSDI reviewers and our shepherd, Ding Yuan, for their valuable comments that improved the paper. We appreciate the support from CloudLab [43] for providing a great research experiment platform. We also thank Yezhuo Zhu for sharing ZooKeeper production traces and Jinfeng Yang for sharing HDFS production traces. This work was supported in part by a Microsoft Azure Research Award.

References

[1] Asana service outage on September 8th, 2016. https://blog.asana.com/2016/09/yesterdays-outage/.

[2] AspectJ, aspect-oriented extension to the Java programming language. https://www.eclipse.org/aspectj.

[3] GoCardless service outage on October 10th, 2017. https://gocardless.com/blog/incident-review-api-and-dashboard-outage-on-10th-october.

[4] Google Compute Engine incident 16007. https://status.cloud.google.com/incident/compute/16007.

[5] gRPC, a high performance, open-source universal RPC framework. https://grpc.io.

[6] Microsoft Azure status history. https://azure.microsoft.com/en-us/status/history.

[7] Protocol buffers. https://developers.google.com/protocol-buffers/.

[8] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, Apr. 2000.

[9] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, Monte Verità, Switzerland, May 2009. USENIX Association.

[10] Amazon. AWS service outage on October 22nd, 2012. https://aws.amazon.com/message/680342.

[11] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS '01. IEEE Computer Society, 2001.

[12] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI '04, San Francisco, CA, 2004. USENIX Association.

[13] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[14] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computing*, 51(5):561–580, May 2002.

[15] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end performance analysis of large-scale Internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 217–231, Broomfield, CO, 2014. USENIX Association.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, Indianapolis, Indiana, USA, 2010. ACM.

[17] J. Dean. Designs, lessons and advice from building large distributed systems, 2009. Keynote at The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS).

[18] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[19] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Santa Clara, California, 2013. ACM.

[20] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computing*, 52(2):99–112, Feb. 2003.

[21] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI '07, Cambridge, MA, 2007. USENIX Association.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, Bolton Landing, NY, USA, 2003. ACM.

[23] E. Gilman. PagerDuty production ZooKeeper service incident in 2014. https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet/.

[24] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210. ACM, 1989.

[25] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 1–14, Oakland, CA, USA, 2018. USENIX Association.

[26] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference*, SIGCOMM '15, pages 139–152, London, United Kingdom, 2015. ACM.

[27] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 175–188, Stevenson, Washington, USA, 2007. ACM.

[28] A. Haeberlen and P. Kuznetsov. The fault detection problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 99–114, Nîmes, France, 2009. Springer-Verlag.

[29] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The φ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 66–78. IEEE Computer Society, 2004.

[30] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, Whistler, BC, Canada, 2017. ACM.

[31] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '10, Boston, MA, 2010. USENIX Association.

[32] R. E. Kalman. On the general theory of control systems. *IRE Transactions on Automatic Control*, 4(3):110–110, December 1959.

[33] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 427–442, Lombard, IL, 2013. USENIX Association.

[34] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the Twenty-third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, Cascais, Portugal, 2011. ACM.

[35] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393, Monterey, California, 2015. ACM.

[36] Microsoft. Office 365 service incident on November 13th, 2013. https://blogs.office.com/2012/11/13/update-on-recent-customer-issues/.

[37] J. C. Mogul, R. Isaacs, and B. Welch. Thinking about availability in large service infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 12–17, Whistler, BC, Canada, 2017. ACM.

[38] D. Nadolny. Network issues can cause cluster to hang due to near-deadlock. https://issues.apache.org/jira/browse/ZOOKEEPER-2201.

[39] D. Nadolny. Debugging distributed systems. In *SREcon 2016*, Santa Clara, CA, Apr. 2016.

[40] Oracle. Java Future and FutureTask. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html.

[41] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[42] J. Postel. DoD Standard Transmission Control Protocol, January 1980. RFC 761.

[43] R. Ricci, E. Eide, and the CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[44] T. Schlossnagle. Monitoring in a DevOps world. *Communications of the ACM*, 61(3):58–61, Feb. 2018.

[45] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[46] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, Mississauga, Ontario, Canada, 1999. IBM Press.

[47] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, The Lake District, United Kingdom, 1998. Springer-Verlag.

[48] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, Banff, Alberta, Canada, 2001. ACM.

[49] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, Broomfield, CO, 2014. USENIX Association.

[50] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 293–306, Hollywood, CA, USA, 2012. USENIX Association.

[51] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the First Conference on Hot Topics in System Dependability*, HotDep '05, Yokohama, Japan, 2005. USENIX Association.