Gray Failure: The Achilles' Heel of Cloud-Scale Systems

Peng Huang[†*] Chuanxiong Guo[†] Lidong Zhou[†] Jacob R. Lorch[†]
Yingnong Dang[‡] Murali Chintalapati[‡] Randolph Yao[‡]

[†]*Microsoft Research* [‡]*Microsoft Azure* [*]*Johns Hopkins University*

Abstract

Cloud scale provides the vast resources necessary to replace failed components, but this is useful only if those failures can be detected. For this reason, the major availability breakdowns and performance anomalies we see in cloud environments tend to be caused by subtle underlying faults, i.e., *gray failure* rather than fail-stop failure. In this paper, we discuss our experiences with gray failure in production cloud-scale systems to show its broad scope and consequences. We also argue that a key feature of gray failure is *differential observability*: that the system's failure detectors may not notice problems even when applications are afflicted by them. This realization leads us to believe that, to best deal with them, we should focus on bridging the gap between different components' perceptions of what constitutes failure.

1 Introduction

The cloud possesses an abundance of redundant components, providing many opportunities to tolerate faults so a system can continue to run. However, to make best use of this ability, a system must be able to rapidly and reliably detect when a component is failing. For this reason, cloud practitioners are frequently challenged by *gray failure*: component failures whose manifestations are fairly subtle and thus defy quick and definitive detection. Examples of gray failure are severe performance degradation, random packet loss, flaky I/O, memory thrashing, capacity pressure, and non-fatal exceptions.

As cloud systems increase in scale and complexity, gray failure becomes more common. Rare events increase in frequency and their effect is amplified by the complex interactions, interference, and dependencies among cloud components executing diverse workloads in a multi-tenancy environment. Our first-hand experience with production cloud systems reveals that gray failure is behind most cloud incidents.

Developers generally follow the common practice of building fault-tolerant and highly available systems by introducing redundancy, failure detection, and failure recovery. But, such mechanisms are inadequate to deal with gray failure, and in some cases even aggravate the situation. They often go wrong by assuming an overly simple failure model in which a component is either correct or stopped (i.e., *fail-stop*), and can be recovered through simple mechanisms such as rebooting. Understanding and defining gray failure despite its variability is thus key to building highly available cloud systems.

In this paper, we discuss in detail the gray-failure problem that remains largely overlooked in the literature. We present real-world examples to understand their characteristics and the risks they pose to cloud systems. Drawing from these data points, we make the first attempt to characterize gray failure.

We find that a key feature that instances of gray failure possess is that they are perceived differently by different entities; we call this *differential observability*. Specifically, one entity is negatively affected by the failure and another entity does not perceive the failure; this is problematic because the latter entity is responsible for failure detection and recovery. For instance, if a system's request-handling module is stuck but its heartbeat module is not, then an error-handling module relying on heartbeats will perceive the system as healthy while a client seeking service will perceive it as failed. As another example, if a link is operating at significantly lower bandwidth than usual, a connectivity test will reveal no problems but an application using the link may obtain bad performance.

One way to deal with gray failure is to *mask* it using protocols robust to gray failure. An example is Byzantine-fault-tolerant (BFT) state machines [6], which tolerate arbitrary faults of fewer than $1/3$ of participating machines and thus tolerate gray failure as a special case. However, Clement et al. [8] show the difficulty of making BFT systems tolerate faults that take the form of slow operation, which is a common way gray failure manifests. Also, BFT has yet to be used in a production system, anecdotally because of its high overhead and complexity. After all, gray failure in most cases is not arbitrary and can be dealt with more efficiently.

A more fundamental problem with masking gray fail-

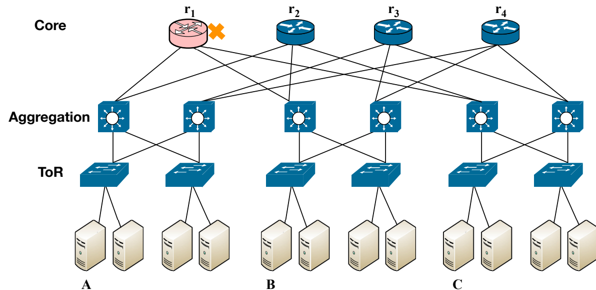

Figure 1: Gray failure in a typical Clos network

ure instead of detecting it is that failed components may not get replaced, leading to their number eventually exceeding the number that can be tolerated. We therefore advocate tackling the problem of gray failure head-on by addressing its fundamental trait, differential observability. We outline potential solutions along this line.

2 Cloud Anomalies with Gray Failure

Our experience with real incidents in Azure, a major cloud service, reveals interesting and somewhat unexpected interplay between gray failure and fault-tolerance mechanisms. This leads to counter-intuitive anomalies; we now highlight a few illuminating cases.

2.1 High redundancy hurts

Cloud data center networks use high redundancy to tolerate failures, typically with a Clos network [1, 4, 12, 22] as illustrated in Figure 1. In such networks, applications are usually unaffected by switches stopping because protocols re-route packets through other redundant paths. For example, if servers A and B are communicating via core switch r_1 but r_1 crashes, packets are re-routed through r_2, r_3, or r_4. Increasing redundancy thus helps availability.

A switch often can also experience intermittent gray failure, e.g., random and silent packet drops [14]. Routing protocols typically do *not* re-route packets in such cases, unlike when switches crash. So, gray failure can result in application glitches or increased latency.

As a consequence, we sometimes see cases where increasing redundancy actually *lowers* availability. For example, consider the following common workload pattern: to process a request, a front-end server must fan out requests to many back-end servers and wait for almost all of them to respond. If there are n core switches, the probability that a certain core switch is traversed by a request is $1-(\frac{n-1}{n})^m$, where m is the fan-out factor. This probability rapidly approaches 100% as m becomes large, meaning each such request has a high probability of involving every core switch. Thus a gray failure at *any* core switch will delay nearly *every* front-end request. Consequently, increasing redundancy can counter-intuitively hurt availability because the more core switches there are, the more likely *at least* one of them will experience a gray failure. This is a classic case where considering gray failure forces us to re-evaluate the common wisdom of how to build highly available systems.

2.2 Under the radar of failure detectors

A typical class of gray failure involves failure detection being so coarse-grained that it does not exercise some important path. For example, we observe incidents where unhealthy VMs are internally experiencing severe network connectivity issues, e.g., due to a driver bug. However, the failure detector, a remote compute manager, does observe not any problems because it does not exercise the VM's external network: it receives VM heartbeats forwarded via a host agent that is able to communicate with the VM via local RPCs. Thus, because of the observational differences between in-VM applications and the failure detector, no recovery happens until a user reports an issue. This creates a long gap between the time when a user is affected and the time when the system becomes aware of the failure (Figure 3). What is missing is a way for the compute manager to observe the VM's internal information.

2.3 Recovery that kills, rather than heals

Azure Storage uses data servers to store data and a storage manager to decide which data servers to store data on. In one instance, a certain data server was experiencing a severe capacity constraint, but a subtle resource-reporting bug caused the storage manager to not detect this gray failure condition. Thus, the storage manager continued routing write requests to this degraded server, causing it to crash and reboot. Of course, the reboot did nothing to fix the underlying problem, so the storage manager once again routed new write requests to it, causing it to crash and reboot again. After a while, a failure detector detected that the data server was repeatedly rebooting, concluded that it was irreparable, and took it out of service. This, along with another subtlety in the replication workflow, reduced the total available storage in the system and put pressure on the remaining healthy servers, causing more servers to degrade and experience the same ultimate fate. Naturally, this eventually led to a catastrophic cascading failure.

2.4 The blame game

The Azure IaaS service provides VMs to its customers using highly complex subsystems including compute,

storage, and network. In particular, VMs run in compute clusters but their virtual disks lie in storage clusters accessed over the network. Even though these subsystems are designed to be fault-tolerant, parts of them occasionally fail. So, occasionally, a storage or network issue makes a VM unable to access its virtual disk, and thus causes the VM to crash. If no failure detector detects the underlying problem with the storage or network, the compute-cluster failure detector may incorrectly attribute the failure to the compute stack in the VM. For this reason, such gray failure is challenging to diagnose and respond to. Indeed, we have encountered cases where teams responsible for different subsystems blame each other for the incidents since no one has clear evidence of the true cause.

3 Modeling and Defining Gray Failure

Although anecdotes of gray failure have been circulating among practitioners and in the literature for years [3, 11, 13, 19, 20], the term gray failure still lacks a precise definition. While it is often associated with performance degradation, intermittent misbehavior, *fail-slow* behavior, or capacity reduction, none of these characteristics capture the true essence of gray failure. We consider modeling and defining gray failure a prerequisite to addressing the problem. After an extensive study of real incidents in Azure cloud services, we make an attempt to define gray failure using the generic model depicted in Figure 2.

3.1 Terminology

Abstractly, we consider two logical entities in our model: a `system`, which provides a service, and an `app`, which uses `system`. Examples of a `system` include a distributed storage service, a data center network, a web search service, and an IaaS platform. An `app` could be a web application, a user, or an operator. One `system` may be an `app` for another `system`. For example, a data center network provides packet transmission service for a storage service. Within a `system`, an `observer` actively or passively gathers information about whether the `system` is failing or not. Based on the observations, a `reactor` takes actions to recover the `system`. The `observer` and `reactor` are considered part of the `system`.

3.2 Differential observability

In addition to the system's internal `observer`, an `app` that uses the `system` also makes its own observations about the health of the `system`. Such observations are typically based on application-specific, end-to-end metrics such as query latency and remote I/O status. Because it is common for a cloud system to be used by different types of `apps`, the `apps`' observations may be derived from different metrics.

We then define gray failure as a form of differential observability. More precisely, a `system` is defined to experience gray failure when *at least* one `app` makes the observation that `system` is unhealthy, but `observer` observes that `system` is healthy.

Table 1 illustrates the spectrum of possible observability cases, in which our definition of gray failure occupies one quadrant. In case ❶, neither the `app` nor the `observer` observes a problem, so there is no failure at all, let alone gray failure. We consider case ❷, in which the `app` observes a failure but the `observer` does not, to constitute gray failure, since users are suffering but the `reactor` will not be invoked to help fix the problem. In case ❸, we also have differential observability, but of the good kind: even though the `app` is not yet experiencing problems, the `observer` knows of some issue and will take proactive steps to repair it. (This case can be problematic if it is a false positive, but that is a different kind of problem than gray failure.) In case ❹, both the `app` and `observer` agree that the `system` is experiencing a failure, so the problem will soon be rectified. Crash and fail-stop failures fall under this case.

Note that gray failure is not unique to large systems. Small-scale or even single-node systems can also experience peculiar failure symptoms. Our above characterization is independent of the scale or nature of a system.

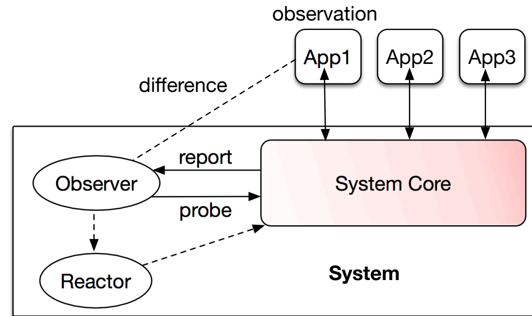

Figure 2: An abstract model to characterize gray failure

	A_i^{good}	A_i^{bad}
S^{good}	❶	❷
S^{bad}	❸	❹

Table 1: Cases induced by the `system`'s observations ($S^{good|bad}$) and its ith `app`'s observations ($A_i^{good|bad}$)

But, we focus our discussion on large distributed systems, where gray failure is prevalent and detrimental to availability.

3.3 Temporal evolution

Gray failure tends to exhibit an interesting evolution pattern along the temporal dimension: initially, the `system` experiences minor faults (latent failure) that it tends to suppress. Gradually, the `system` transits into a degraded mode (gray failure) that is externally visible but which the `observer` does not see. Eventually, the degradation may reach a point that takes the `system` down (complete failure), at which point the `observer` also realizes the problem. A typical example is a memory leak.

We can use the differential observability trait and our model to characterize this pattern: when the observations of both the `observer` and the `app` are good, the `system` is either working well or having minor latent failure; as soon as *at least* one `app` observes that the `system` is not doing well while the `observer` still perceives the `system` as healthy, gray failure starts to happen. Eventually, the `observer` also detects the problem so the differential observability disappears, but by this point the gray failure has already evolved into a complete failure. This process manifests as a transition from ❶ to ❷ to ❹ in Table 1. During gray failure with intermittent misbehavior, this type of transition happens repeatedly.

3.4 Applying the model

We now apply our model to two gray failure cases and anomalies described earlier (§2). In the network case (§2.1), some service (the `app`) transmits packets through the network (the `system`). The switches are the `observers` and the routing protocol is the `reactor`. If a core switch experiences random packet drops, its neighbor `observers` will not perceive it as failed and thus packets will not be re-routed. However, if the `app` has a high-fan-out workload pattern, it is likely to observe a problem even though the `observers` do not, leading to differential observability and gray failure.

In the storage service example (§2.3), when several data servers are under severe capacity pressure, the storage manager (acting as both `observer` and `reactor`) is unaware of the issue. But, some VMs (the `apps`) experience remote I/O exceptions, i.e., they observe unhealthiness. This differential observability between `observer` and `app` constitutes gray failure. After the data server crashes, the `observer` detects the failure (the observation difference being temporarily gone) and perceives it as a regular crash. So, the `reactor` makes the data node reboot, making the data node perceived as healthy again. In this vicious loop, more observation differences and instances of gray failure arise. Eventually the system becomes aware of the cascading failures, so the observation differences permanently disappear. But, it is too late.

4 Discussion

The ambiguous nature and temporal idiosyncrasy of gray failure make it distinctly different from what is assumed in typical failure models. This defeats traditional fault-tolerance solutions and thus poses significant challenges to cloud practitioners. The model that we developed in §3 not only provides a definition of gray failure; it also implies a potential solution space. In this section, we outline future directions for addressing gray failure.

4.1 Closing the observation gap

A natural solution to gray failure is to close the observation gaps between the `system` and the `apps` that it services. In particular, system `observers` have traditionally focused on gathering information reliably about whether components are up or down. But, gray failure makes these not just simple black-or-white judgments. Therefore, we advocate moving from singular failure detection (e.g., with heartbeats) to multi-dimensional health monitoring. This is analogous to making assessments of a human body's condition: we need to monitor not only his heartbeat, but also other vital signs including temperature and blood pressure. For example, to address the problem of invisible VM connectivity issues from §2.2, we could leverage in-VM performance counters to detect connectivity issues earlier. We could thereby avoid customer-reported incidents and lengthy troubleshooting (Figure 3).

4.2 Approximating application views

Although it would be ideal to eliminate differential observability completely by letting the `system` measure what its `apps` observe, it is practically infeasible. In a multi-tenant cloud system that supports various applications and different workloads, it is unreasonable for a system to track how it is used by *all* applications. Also, the modularity principle precludes applications from directly observing the health of internal system components. These constraints imply that observation differences will persist to a certain extent.

One feasible approach is for a `system` to measure metrics that *approximate* the observations of its `apps`. For example, to tackle the network gray failure example (§2.1), the cloud system can send probes to measure server-to-server latency and reachability to emulate observations of the network by common applications, as in

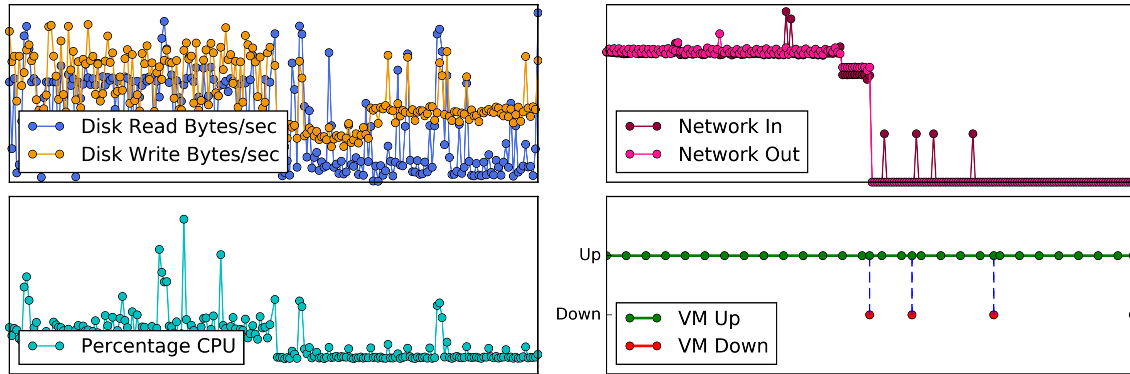

Figure 3: The first three graphs show various in-VM performance counters before and during an instance of gray failure (§2.2); the network counters drop shortly after the failure-triggering event (VM migration). The last graph, in the bottom right, shows the compute manager's observations over time: green dots represent a belief that the VM is healthy and red dots represent a belief that it is unhealthy. Each red dot here is caused by a user-initiated reboot; the underlying issue is not detected until well after the time period depicted here.

Pingmesh [14]. Such approximation can significantly reduce the chance of gray failure due to differential observability. However, solutions along this line must address the challenge that overly active probing may further burden an already degraded system and negatively impact system health. Recall that an `app` is a logical entity that could represent another `system` (§3.1), so approximating application views is not necessarily the same as taking an end-to-end approach.

4.3 Leveraging the power of scale

While the scale of cloud systems contributes to the rising frequency of gray failure, we can also *leverage* this scale to tackle the challenge of gray failure. In particular, since gray failure is often due to isolated observations of an `observer`, leveraging the observations from a large number of different components that are complementary to each other can help uncover gray failure rapidly. Indeed, many gray failure cases we investigated are only detectable in a distributed fashion because each individual component has only a partial view of the entire system (so the gray failures are intrinsic). Even for cases where the underlying problem is simply that the `observer` is doing a poor job of detecting failures (so the gray failures are extrinsic and could be avoided by fixing the `observer`), such distributed observation can also be helpful.

Where to conduct such aggregation and inference is an interesting design question to explore. If it is done too close to the core of a `system`, it may limit what can be observed. If it is near the `apps`, the built-in system fault-tolerance mechanisms that try to mask faults may cause differential observability to be exposed too late. We envision an independent plane that is outside the boundaries of the core `system` but nevertheless connected to the `observer` or `reactor`.

Sometimes, a gray failure that is observable by an `app` may not be readily *detectable*. For example, a random-packet-drop gray failure results in differential observability between the `system` and `app`, but it is often unclear *which* network device is responsible for the difference. Investigating an individual case is particularly difficult due to the transient nature of the issue and the lack of information. But, perhaps with the help of global-scale probing from many devices, we can obtain enough data points to apply statistical inference and thereby identify components with persistent but occasional failures.

There are similar benefits in leveraging scale to address the "blame game" described in §2.4. For instance, we can aggregate observations of VM virtual disk failure events and map them to cluster and network topology information. Indeed, we have used this approach to pinpoint many gray failure cases due to storage overloading or unplanned top-of-rack (ToR) switch reboots. In general, from our experience, we believe that leveraging global data at scale is a promising approach that should be embraced in the battle against gray failure.

4.4 Harnessing temporal patterns

In addition to leveraging the spatial dimension, understanding the evolution of gray failure manifestations over time can also help provide early warnings to mitigate them before they have catastrophic impact. As discussed in §3, the prelude to gray failure is usually a latent fault that is too minor for the `observer` to declare a failure. Finding the temporal patterns that lead to gray failure

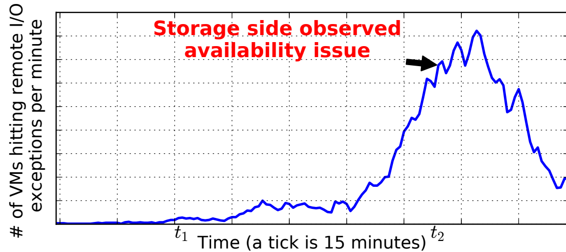

Figure 4: Compute's view of storage service health during an incident involving gray failure (§2.3)

would allow the `system` to react early even before `apps` are affected. But, since most latent faults are benign, this has to be done with care so as not to trigger too many false positives.

Even after a gray failure strikes, there remains an opportunity to react before it leads to a widespread complete failure. For example, in the storage gray failure case in §2.3, which later leads to a large-scale cascading failure, Figure 4 shows two time points t_1 and t_2. Here, t_1 is when the compute service observes an uptick in remote I/O exceptions. But, it is not until t_2 that the storage service detects the issue and takes action. Thus, there is a window of opportunity between t_1 and t_2 to prevent the cascading failure by harnessing and correlating differential observability over time.

5 Related Work

The phenomenon of gray failure is not new, so unsurprisingly there has been literature and system-incident reporting [3, 19] mentioning the symptoms of gray failure. For instance, Gray [11] discusses the Heisenbug—a bug that seems to disappear when one attempts to study it—and proposes techniques such as re-execution to deal with it. Gunawi et al. [13] analyze public cloud service outage reports and show cases that we classify as gray failure. An internal study of failures in a major cloud service by Huang et al. [15] discusses *fail-slow* faults. However, none of these works focus on gray failure or attempt to define the problem domain.

A myriad of techniques have been proposed to leverage redundancy and replication to tolerate component faults, e.g., primary/backup replication [2], RAID [21], Paxos [16], and chain replication [23]. Many of these techniques assume a simple failure model: fail-stop. Different from fail-stop, a component experiencing gray failure appears to be still working but is in fact experiencing severe issues. Such discrepancy can negatively impact traditional techniques and cause fault-tolerance anomalies (§2). Even techniques that are designed to mask *Byzantine* faults, as discussed in §1, can be insufficient or inefficient for gray failure.

Several works have proposed ways to improve failure detectors in asynchronous, distributed environments. Falcon [18] leverages a network of *spies* across layers of a system to make failure detection more reliable. Pigeon [17] exposes uncertain failure information to applications to allow applications to take informed recovery actions. However, all these reliable failure detectors are designed for fail-stop failures; to alleviate the gray failure problem, we argue that the system needs to go beyond traditional failure detection to health monitoring. The differential observability trait that we describe can be leveraged to reliably detect gray failure.

Much work has been done to collect system performance metrics and then apply statistical techniques to diagnose or detect performance issues [5, 7, 9, 10]. For example, Cohen et al. [9] proposes Tree Augmented Naive Bayesian networks to correlate low-level performance metrics with high-level Service Level Objectives (SLO). While these works can help address certain types of gray failure, they tend to look at the system by itself and conduct analysis reactively. Moreover, there are many types of gray failure that are not performance-related. We advocate leveraging diverse observations from different entities to complete their health views and proactively enhance failure handling.

6 Conclusion

As cloud systems continue to scale, the overlooked gray failure problem becomes an acute pain in achieving high availability. Understanding this problem domain is thus of paramount importance. Drawing from our experiences with a major cloud service, we discuss the gray failure problem and make the first attempt to define it. We argue that to address the problem, it is crucial to reduce *differential observability*, a key trait of gray failure.

Acknowledgments

We would like to thank the anonymous reviewers and workshop attendees for their feedback, which has greatly improved our paper. We also thank Microsoft Azure for providing support, sharing gray-failure war stories, and participating in heated discussions.

References

[1] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *Proceedings of the 2008 ACM SIGCOMM Conference* (Aug. 2008), pp. 63–74.

[2] ALSBERG, P. A., AND DAY, J. D. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd Interna-*

tional Conference on Software Engineering (ICSE) (Oct. 1976), pp. 562–570.

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

[4] ANDREYEV, A. Introducing Data Center Fabric, The Next-generation Facebook Data Center Network. https://code.facebook.com/posts/360346274145943/, Nov. 2014.

[5] BODIK, P., GOLDSZMIDT, M., FOX, A., WOODARD, D. B., AND ANDERSEN, H. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (Apr. 2010), pp. 111–124.

[6] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)* (Feb. 1999), pp. 173–186.

[7] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. 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)* (Oct. 2014), pp. 217–231.

[8] CLEMENT, A., WONG, E., ALVISI, L., DAHLIN, M., AND MARCHETTI, M. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2009), pp. 153–168.

[9] COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J., AND CHASE, J. S. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)* (2004), pp. 16–16.

[10] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2005), pp. 105–118.

[11] GRAY, J. Why do computers stop and what can be done about it? In *Proc. Symposium on Reliability in Distributed Software and Database Systems* (1986), pp. 3–12.

[12] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A scalable and flexible data center network. In *Proceedings of the 2009 ACM SIGCOMM Conference* (Aug. 2009), pp. 51–62.

[13] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)* (Oct. 2016), pp. 1–16.

[14] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 139–152.

[15] HUANG, P., JIN, X., BOLOSKY, W. J., AND ZHOU, Y. Why does a cloud-scale service fail despite fault-tolerance? *Unpublished internal document* (2014).

[16] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (May 1998), 133–169.

[17] LENERS, J. B., GUPTA, T., AGUILERA, M. K., AND WALFISH, M. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)* (Apr. 2013), pp. 427–442.

[18] LENERS, J. B., WU, H., HUNG, W.-L., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2011), pp. 279–294.

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

[20] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS)* (Mar. 2003).

[21] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (1988), pp. 109–116.

[22] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *Proceedings of the 2015 ACM SIGCOMM Conference* (Aug. 2015), SIGCOMM '15, pp. 183–197.

[23] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design (OSDI)* (Dec. 2004), pp. 91–104.