# Failure Recovery: When the Cure Is Worse Than the Disease

Zhenyu Guo[†] Sean McDirmid[†] Mao Yang[†] Li Zhuang[†] Pu Zhang[†‡] Yingwei Luo[‡]
Tom Bergan[†§] Peter Bodik[†] Madan Musuvathi[†] Zheng Zhang[†] Lidong Zhou[†]
[†]*Microsoft Research* [‡]*Peking University* [§]*University of Washington*

## ABSTRACT

Cloud services inevitably fail: machines lose power, networks become disconnected, pesky software bugs cause sporadic crashes, and so on. Unfortunately, failure recovery itself is often faulty; e.g. recovery can accidentally recursively replicate small failures to other machines until the entire cloud service fails in a catastrophic outage, amplifying a small cold into a contagious deadly plague! We propose that failure recovery should be engineered foremost according to the maxim of **primum non nocere**, that it "does no harm." Accordingly, we must consider the system holistically when failure occurs and recover only when observed activity safely allows for it.

## 1 FAILURES OF FAILURE RECOVERY

[27] **Leap day 2012, Microsoft's Azure** cloud experienced a serious outage when a small number of servers were killed due to a virtual machine initialization bug that was incorrectly attributed to hosting servers. Failure recovery then reincarnated a dead server's virtual machines to other healthy servers, unwittingly replicating the bug across the cluster *ad infinitum*.

[4] **April 2011, Amazon's EC2** experienced a disruption when, after a network outage, data store nodes recovered from the seeming death of their replicas by requesting storage to create new ones. However, the still-alive replicas held onto data, and hence storage, until these new replicas were ready, causing storage exhaustion.

[13] **September 2010, Facebook** disconnected many users when an invalid configuration value in a database caused clients to recover by re-requesting it, quickly overwhelming the database. Failed database queries themselves then caused invalidation of good configuration values retrieved after the database was fixed, causing a vicious feedback loop only fixed by a complete reboot.

[17] **September 2009, Google's** Gmail suffered a widespread outage when a few Gmail servers were taken offline for routine maintenance. Load was underestimated after a recent upgrade so affected request routers became overloaded. Recovery pushed traffic to other request routers, causing them to become overloaded themselves. Within minutes all routers were overloaded.

## 2 PRIMUM NON NOCERE

In the previous cases, the cure of failure recovery was worse than the diseases of failure being treated. Although these cases might seem exceptional, our investigation of major outages in the last few years (including [3–5, 13, 17, 18, 27, 32]) reveals that "death by failure recovery" is a common problem: recovery mechanisms often amplify and prolong failures rather than resolve them. Much work has already explored common problems in distributed systems [6, 14, 19–21, 28, 29] and improving failure recovery (e.g. [25, 30]); this paper instead focuses on a much less discussed topic: how can failure recovery avoid making things worse? Failure recovery should adhere to a modified version of primum non nocere:

> *Given an existing failure, it may be better not to do something, or even to do nothing, than to accidentally risk inducing catastrophic failure.*

Recovery from failure is just secondary to the goal of not killing the system. Each event in Section 1 violated this maxim: Microsoft [27] replicated a machine-killing bug to many machines; Amazon [4] exhausted storage via catch-22 replica allocation; Facebook [13] clients overwhelmed a databases with pointless configuration value refresh requests; and Google [17] request routers were recursively overloaded all. On the other hand, no failure recovery at all would mean tolerating rather than treating failures–some virtual machine work would be dropped, data would be inaccessible, configuration values would be incorrect, and mailboxes would be unviewable–but at least the systems would continue to work.

Should we seriously consider "do nothing" failure recovery, i.e. a *placebo*? Probably not: we prefer that our systems actually recover from failures to minimize service degradation and maximize availability. The rest of this paper explores when failure recovery goes wrong and how it can be safer with respect to "do no harm."

## 3 SYSTEMS THINKING

We often design failure recovery to react to specific failure events in specific parts of a system. However, these designs lead to unanticipated interactions with the rest of the system that can result in adverse consequences; e.g. recovery after correlated failures can easily exhaust

system resources, causing more failures. We observe that safer failure recovery must emphasize *systems thinking* [11] to consider the context of failure recovery's relationship and impact on the system as a whole, focusing on cyclical, rather than linear, cause and effect. The rest of this section describes significant classes of malignant failure recovery along with strategies for solving these problems that involve coarsely assessing non-local system conditions before deciding if recovery is reasonable.

## Resource Contention

Failure recovery often consumes a significant amount of resources such as CPU time to access a database, network bandwidth to read or write a replica, and storage space to replicate data. Consuming these resources is benign if systems are provisioned with enough extra resources in support of failure recovery. Unfortunately, single events often trigger many concurrent failures; e.g., nodes share software bugs or bad data or, after a power outage, regain power at the same time. Nodes recovering from the same failure can easily consume the same resources to the point of exhaustion given that the same recovery code executes to recover from these failures.

Consider how in April 2011, Amazon's EC2 (Elastic Compute Cloud) experienced a significant service disruption [4] due to a network outage where data store nodes lost connections to their replicas and mistakenly assumed that they had died. When the network was restored, these nodes began recovering by rapidly requesting disk space to create new replicas. Storage capacity quickly became exhausted given many nodes looking for new storage, coupled with the unfortunate precaution that the old replicas, actually still alive, were required to hold onto replicated data until new replicas replaced them. Many nodes then became stuck waiting for free space while clients became stuck waiting for the nodes.

Failure detection often consists of liveness checks that misdiagnose unresponsive machines as failed because they are contending for resources. A Microsoft data center in March 2011 experienced an outage due to a bug where jobs were not deleted from a persistent queue after being completed by a databus service. After the databus service was upgraded, many threads were created to concurrently process all of these stale jobs, quickly overloading the servers, which were then detected as dead due to their unresponsiveness. Other servers began recovery by replicating data that was considered lost, but were soon detected as dead since the large amount of replication overloaded them.

In the above examples, failure recovery caused extraordinary resource contention within the system. Beyond provisioning and preserving resources for failure recovery, usage can be throttled to avoid demand spikes, i.e. "slower is faster." In both the Amazon and Microsoft cases, concurrent replications could be restricted in number so that progress in recovery could occur if albeit more slowly. Also in the Microsoft case, failure detection was falsely triggered, which could be alleviated by isolating the resource usage of failure detection.

## Misguided Admission Control

Admission control recovers from service overload by preventing the processing of new requests. Although this mechanism has been deeply studied [21, 28], we found that admission control itself can cause catastrophic failure by propagating overloading conditions in a globally overloaded system where rebalancing is not effective.

In September 2009, Google's Gmail suffered a widespread outage [17] when a few servers were taken offline for routine maintenance. Unfortunately, load was underestimated after a recent upgrade, causing affected request routers to become overloaded. Recovery mechanism redirected all traffic to other request routers, which then became overloaded themselves. All request routers eventually became overloaded, causing an outage until more capacity was added.

As another example, consider a Skype outage [32] in December of 2010 when 20% of all Skype clients crashed because of a software bug. These crashes, not catastrophic in themselves, affected 25%–30% of all clients that acted as "supernodes" by providing directory services. Given the many failed supernodes combined with many crashed clients restarting and accessing directory services, the remaining supernodes became overloaded and, as a form of admission control, were shutdown to protect their hosting machines—the system was trapped in a catastrophic feedback loop.

Although admission control policies are often correct locally, they are less robust within the context of the entire system that can be entirely overloaded. In this case, admission control should not introduce a drastic phase change in traffic volumes for servers. Rather than deny admission completely, admission control should steady a server's processing rate, and deny the incoming traffic that is beyond the capacity of the server.

## Misidentified Failure Scope

Failure recovery can unintentionally amplify failures by *misidentifying* the cause, or scope, of a failure and taking unnecessary recovery action. In many cases, recovery action leads to the same failure occurring again, creating a vicious feedback cycle between failure and recovery ac-

tions that quickly brings the system to its knees. Recovery actions that replicate or reincarnate jobs are particularly dangerous because they provide a vector for *failure contagion*; i.e. spreading a failure throughout the system.

Consider Microsofts Azure 2012 leap day outage [27] that was caused by a bug that prevented virtual machine initialization. While not fatal in itself, the system incorrectly attributed the bug to the virtual machines hosting server, killing it and all virtual machines on it. Failure recovery then reincarnated these virtual machines to other healthy servers, unintentionally replicating the same bug to these servers, causing more server shutdowns and more reincarnations to healthy servers, ad infinitum until catastrophic failure occurred.

In September 2010, Facebook suffered a serious outage [13]. An automated system fixes invalid configuration values in a cache by replacing them with fresh values from a database. In this case, however, the configuration value in the database was itself invalid, causing clients to continuously recover by requesting it from the database, which quickly became overwhelmed. Worse still, the unresponsive database was detected as failed by clients, causing continued cache invalidation and further recovery even as the value had already been manually fixed in the database. The system entered a vicious feedback loop that was only stopped through a complete shutdown.

Failure misidentifications are bugs in failure recovery logic that are difficult to detect and fix proactively because they arise from difficult-to-anticipate or unusual failure scenarios. Beyond more carefully identifying the cause and scope of failures, failure recovery should also be aware of how failures were identified so positive feedback cycles can be detected. When a feedback cycle is detected, failure recovery can decide that an intended recovery action is doing more harm than good, and instead do nothing. Such an action does not resolve the real failure, but it frees up resources needed to resolve the failure manually without taking down entire systems.

# 4 Other Recovery Problems

The following misbehaviors are not direct cases of failed failure recovery, but are related in adverse effects.

## "Recovered" Software Bugs

Failure recovery can hide rare software bugs as a side affect so that the system can continue working. This is good for maintaining the system's availability but can hide bugs from developers. Take the Amazon case [4] above as an example: when a large number of replica groups were aggressively searching for storage space to restore "stuck" replicas, a race condition was triggered causing even more nodes to become stuck, leading to an eventual brownout. It is likely that this race condition had manifested before but was masked by failure recovery.

As another example, a Google Paxos cluster was misconfigured such that one of the replica could never join the replication group successfully because its identity was misspelled in the config file [8]. Paxos can tolerate failures from less than majority of group members, which therefore masked this configuration error. However, the system's resilience to failure is degraded from two server failures to only one, making it less reliable.

As shown in the above cases, failure recovery masks software bugs, making systems more fragile during failure recovery. Even worse, these bugs often surface when the system is in a critical state. It is then unwise to cover up these bugs using failure recovery, and we advocate that an audit mechanism should be present to check whether the recovered failure is previously known or not.

## Service Dependency

A dependency between two subsystems can result in failure amplification when a failure in one propagates to the other. The dependencies can either be explicitly designed or inadvertently introduced due to hidden resource sharing between otherwise independent services.

Many popular distributed database services are built on top of replicated file systems: consider Google's BigTable [10] on top of the Google File System (GFS) [10]; Amazon's Replicated Database Service (RDS) [2] on top of the Elastic Block Service (EBS) [1]; and Microsoft Azure's table abstraction on top of its stream layer [7]. Although such layering simplifies handling of failure in the upper layer, it does not consider the intricate interplay between layers during failure. In the above Amazon case [4], a large number (13%) of the underlying EBS services were down due to a network misconfiguration. However, there was a much bigger portion of the RDS population (e.g., 41%) that was "stuck" in IO than the corresponding failed EBS volume population. This occurred because one RDS instance makes use of multiple EBS volumes, meaning that failure in any of the participating EBS volumes can make the RDS unavailable. Ironically, one goal of making use of multiple EBS volumes should be fast failure recovery as the system can copy the state from multiple places simultaneously, which however in this case amplifies failure.

The same Amazon case above also provides an example of a failure propagation between seemingly independent services. RDS and EBS clusters belonging to different availability zones (AZ) used the same control plane; failures in one AZ quickly saturated request queues in the

shared control plane. This blocked requests to other AZs, in effect propagating the failure across fault domains.

One way to alleviate this problem is to remove and degrade service dependencies when possible; e.g. the recovery of the replication groups in replicated file systems can be autonomous using Paxos [24] instead of counting on a meta service as a commanding brain. When a dependency is indispensable, service dependencies must be managed carefully to avoid any potential failure amplification. For example, in a layered design, small set of machines can be organized into "containers," where the instances of the upper layer in a container depend only on the instances of the lower layer in the same container. This kind of alignment allows for a constrained, yet still effective, parallel recovery, while at the same time preventing failure from propagating beyond container boundaries. Similarly, when multiple services are sharing the same resource, such as a request queue, carefully-designed admission-control policies should isolate one service from excessive resource usage by the other.

## 5 Towards Safe Failure Recovery

Section 3 discussed how common failure recovery problems can be mitigated by reacting more conservatively according to global system contexts. Unfortunately, this insight is not a panacea: as a distributed system is very complex, how could failure recovery ever know enough about its system context to decide if it is doing more harm than good? We close this paper with potential research directions that could help ensure more safety through automatic detection of failure recovery problems.

### Global Reasoning with Abstract Models

As failure amplification requires global system reasoning, one possible approach is to build and analyze abstract models of the whole system. We first explore solving this problem with *system dynamics* [15] that aims to understand complex system behavior over time by dealing with internal feedback loops and time delays that affect an entire system. Figure 1 shows a graph for a simplified model of system dynamics for Section 3's Skype case [32] that demonstrates the consequences of misguided admission control. The graph is a causal loop diagram where vertices are key variables in the system and edges represent feedback between two connected variables. When the number of live Skype supernodes decreases, the number of disconnected regular nodes increases (minus "–" sign edges). Similarly, when the number of disconnected regular nodes increases, the number of regular nodes that are trying to connect to supernodes also increases (plus "+" sign edges). The admission control policy is depicted as two outgoing edges from `Connected Regular Node#`: when the number of the regular nodes connected to a supernode exceeds a `THRESHOLD`, the supernode shuts down and so decreases the number of live supernodes; when the threshold is not exceeded, the regular nodes connect successfully therefore the number of disconnected regular nodes decreases. We then find two `A` and `B` feedback loops in the graph. Feedback loops with an odd count of – transitions [23] are negative that result in a stable system, while loops with even count represent self-reinforcing processes that can destabilize the system. When the system triggers the `B` feedback loop, the number of live supernodes can eventually approach zero, causing service outage.

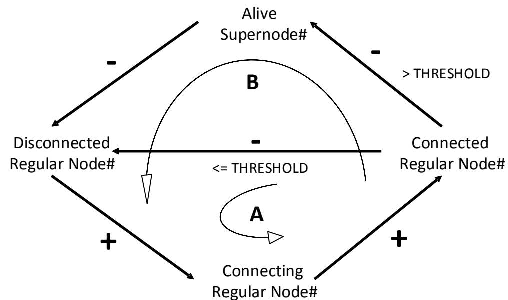


**Figure 1**: A simplified Skype model using System Dynamics.

Our experience reveals several challenges with this abstract modeling approach. (i) The variables required in the model are not directly present in the system design documents or implementations as the latter usually focuses on single node behavior, while the former requires variables that represent global system state. (ii) Precise analysis of the model requires quantitative relationships between key variables. Moreover, some of these relationships involve sets of discrete values; e.g. replication group membership in the Amazon EC2 case. One possible approach is to extend system dynamics with Petri Net [16]. (iii) The system contains multiple behavioral phases represented by different relationships between system components. A simple example is a request queue filling up on load, resulting in increased latency, but once full, the queue starts dropping requests. (iv) The model must explicitly define what is failed failure recovery. Although easy for System Dynamics (as positive feedback loops), it is difficult for others like Petri Nets.

Finally, inferring a model automatically and keeping it consistent as the system evolves is challenging. Our initial effort at automatically inferring a model through disciplined intervention experiments [12] pro-

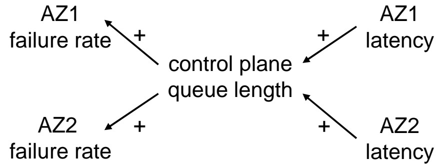


**Figure 2**: Automatically inferred model of failure propagation across different availability zones.

vided promising results. Figure 2 shows the inferred model in a failure scenario similar to the Amazon EC2 failure on our own distributed storage system. For instance, by injecting artificial latency to requests in availability zone (AZ) 1 in Figure 2, we are able to infer that the control plane queue length increases. Surprisingly, we also discovered that such intervention produces extraneous "Heisenberg" dependencies; e.g. by increasing thread utilization in a machine artificially, we inadvertently reduced the utilization of downstream services as the slow machine sent less requests. Pruning such false dependencies in the inferred model remains challenging.

### Testing and Model Checking

Runtime testing is limited because it is always incomplete and the system is not under failure recovery's control. As a result, some failure scenarios cannot be produced as expected even with failure injection techniques. Implementation-level model checking such as MoDist [33] addresses this problem by controlling the whole environment (e.g. scheduling, network message lost, and ordering) through an extra layer that intercepts system calls between the distributed system and the underlying operating system. Our experience productizing MoDist reveals that real systems may use undocumented APIs or shared memory to boost system performance, imposing many challenges for using MoDist. However, for many popular cloud services such as distributed storage systems and large-scale, data-parallel computation engines, such complexity is unnecessary. In fact, these systems can easily be built on top of a small programming model that includes only network messaging, disk IO, thread pools, locks, and timers; see the Tribble [22] project that implements many of these systems.

To further support model checking for failed failure recovery, our experience indicates that the challenges lie in the fidelity of: (i) injected failures, especially correlated failures occurring on multiple nodes; (ii) simulated high workload. We adopt simulated "symbolic" workloads to avoid state space explosions, and our prototype is then feasible due to a small programming model.

### Runtime Global Monitoring

The proposals in Section 3 require monitoring global system state to detect and prevent outage. However, such monitoring is distributed and fragile to common distributed system failures. Our past experience with online distributed system checkers like $D^3S$ [26] suggests the following challenges to address: (i) consistency between the world learned by the target system and the world learned by the monitoring facility, including both the single node state (e.g. if a node is dead) and the state across multiple nodes (e.g. if the invariance between variable `A` on node `X` and variable `B` on node `Y` holds); (ii) realtime state report for quick decision making even when the system is overloaded; (iii) fault-tolerance.

Existing failure detection mechanisms in the target system can be reused to ensure local node state consistency; state reports can be attached to failure detection messages to achieve timeliness; distributed consistent cut algorithms, e.g. Chandy-Lamport [9] as adopted in $D^3S$, can guarantee global state consistency; and streaming engines [31] can aid with reliability. Recent work by Pigeon [25] provide a good starting point by echoing these challenges of accuracy, timeliness, and coverage.

## 6 Conclusion

Failure will occur and so failure recovery is an integral part of a fault-tolerant cloud service. Unfortunately, failure recovery can cause more problems than it solves, and so must be engineered explicitly according to a "do no harm" requirement. Our work focuses and classifies various failure recovery misbehaviors, and proposes using more global reasoning about the system as opposed to local reasoning about the failure. Preventing failure recovery from going bad will become a more important research topic as service availability becomes a point that cloud service providers are increasingly judged on.

## References

[1] Amazon. Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/.

[2] Amazon. Amazon Relational Database Service (Amazon RDS). http://aws.amazon.com/rds/.

[3] Amazon. Summary of the Amazon EC2, Amazon EBS, and Amazon RDS Service Event in the EU West Region. http://aws.amazon.com/message/2329B7/, August 2011.

[4] Amazon. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648/, April 2011.

[5] Amazon. Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region.

http://aws.amazon.com/message/680587/, December 2012.

[6] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4), 2001.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proc. of SOSP*. ACM, 2011.

[8] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[9] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1), 1985.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. OSDI*, 2006.

[11] P. Checkland. *Systems Thinking, Systems Practice*. John Wiley & Sons, 1999.

[12] F. Eberhardt, P. O. Hoyer, and R. Scheines. Combining experiments to discover linear cyclic models with latent variables. *Journal of Machine Learning Research - Proceedings Track*, 9:185–192, 2010.

[13] Facebook. More details on today's outage. http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919, September 2010.

[14] B. Ford. Icebergs in the clouds: the other risks of cloud computing. *CoRR*, 2012.

[15] J. Forrester. Counterintuitive behavior of social systems. In *Technology Review*, 1971.

[16] C. Girault and W. Reisig, editors. *Application and Theory of Petri Nets*, volume 52 of *Informatik-Fachberichte*. Springer, 1982.

[17] Google. More on today's Gmail issue. http://gmailblog.blogspot.com/2009/09/more-on-todays-gmail-issue.html, September 2009.

[18] Google. About today's App Engine outage. http://googleappengine.blogspot.jp/2012/10/about-todays-app-engine-outage.html, October 2012.

[19] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[20] J. Gray. A census of tandem system availability. In *IEEE Transactions on Reliability*, 1990.

[21] S. D. Gribble. Robustness in complex systems. In *HotOS*, 2001.

[22] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Paxos made parallel. Technical Report MSR-TR-2012-118, 2012.

[23] C. W. Kirkwood. System Dynamics Methods: A Quick Introduction. http://www.public.asu.edu/ kirkwood/sysdyn/SDIntro/SDIntro.htm.

[24] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[25] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proc. of NSDI*, 2013.

[26] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging Deployed Distributed Systems. In *Proc. of NSDI*, 2008.

[27] Microsoft. Summary of Windows Azure Service Disruption on Feb 29th, 2012. http://blogs.msdn.com/b/windowsazure/archive/-2012/03/09/summary-of-windows-azure-service-disruption-on-feb-29th-2012.aspx, February 2012.

[28] J. C. Mogul. Emergent (mis)behavior vs. complex software systems. In *EuroSys*. ACM, 2006.

[29] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[30] S. Pertet and P. Narasimhan. Handling cascading failures: the case for topology-aware fault-tolerance. In *HotDep*, 2005.

[31] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Eurosys*, 2013.

[32] Skype. CIO update: Post-mortem on the Skype outage. http://blogs.skype.com/en/2010/12/cio_update.html, December 2010.

[33] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent model checking of unmodified distributed systems. In *Proc. of NSDI*, 2009.