acmqueue

Current Issue Past Issues Topics

October 26, 2017

Volume 15, issue 5

PDF

Abstracting the Geniuses Away from Failure Testing

Ordinary users need tools that automate the selection of custom-tailored faults to inject.

Peter Alvaro and Severine Tymon

The heterogeneity, complexity, and scale of cloud applications make verification of their fault tolerance properties challenging. Companies are moving away from formal methods and toward large-scale testing in which components are deliberately compromised to identify weaknesses in the software. For example, techniques such as Jepsen apply fault injection testing to distributed data stores, and Chaos Engineering performs fault injection experiments on production systems, often on live traffic. Both approaches have captured the attention of industry and academia alike.

Unfortunately, the search space of distinct fault combinations that an infrastructure can test is intractable. Existing failure-testing solutions require skilled and intelligent users who can supply the faults to inject. These superusers, known as Chaos Engineers and Jepsen experts, must study the systems under test, observe system executions, and then formulate hypotheses about which faults are most likely to expose real system-design flaws. This approach is fundamentally unscalable and unprincipled. It relies on the superuser's ability to interpret how a distributed system employs redundancy to mask or ameliorate faults and,

moreover, the ability to recognize the insufficiencies in those redundancies—in other words, human genius.

This article presents a call to arms for the distributed systems research community to improve the state of the art in fault tolerance testing. Ordinary users need tools that automate the selection of custom-tailored faults to inject. We conjecture that the process by which superusers select experiments—observing executions, constructing models of system redundancy, and identifying weaknesses in the models—can be effectively modeled in software. The article describes a prototype validating this conjecture, presents early results from the lab and the field, and identifies new research directions that can make this vision a reality.

The Future is Disorder

Providing an "always-on" experience for users and customers means that distributed software must be *fault tolerant*—that is to say, it must be written to anticipate, detect, and either mask or gracefully handle the effects of fault events such as hardware failures and network partitions. Writing fault-tolerant software—whether for distributed data management systems involving the interaction of a handful of physical machines, or for web applications involving the cooperation of tens of thousands—remains extremely difficult. While the state of the art in verification and program analysis continues to evolve in the academic world, the industry is moving very much in the opposite direction: away from formal methods (with some noteworthy exceptions, however[41]) and toward approaches that combine testing with fault injection.

The following section describes the underlying causes of this trend, why it has been successful so far, and why it is doomed to fail in its current practice.

The Old Gods

The ancient myth: leave it to the experts.

Once upon a time, distributed systems researchers and practitioners were confident that the responsibility for addressing the problem of fault tolerance could be relegated to a small priesthood of experts. Protocols for failure detection, recovery, reliable communication,

consensus, and replication could be implemented once and hidden away in libraries, ready for use by the layfolk.

This has been a reasonable dream. After all, abstraction is the best tool for overcoming complexity in computer science, and composing reliable systems from unreliable components is fundamental to classical system design.[33] Reliability techniques such as process pairs[18] and RAID[45] demonstrate that partial failure can, in certain cases, be handled at the lowest levels of a system and successfully masked from applications.

Unfortunately, these approaches rely on failure detection. Perfect failure detectors are impossible to implement in a distributed system,[9,15] in which it is impossible to distinguish between delay and failure. Attempts to mask the fundamental uncertainty arising from partial failure in a distributed system—for example, RPC (remote procedure calls[8]) and NFS (network file system[49])—have met (famously) with difficulties. Despite the broad consensus that these attempts are failed abstractions,[28] in the absence of better abstractions, people continue to rely on them to the consternation of developers, operators, and users.

In a distributed system—that is, a system of loosely coupled components interacting via messages—the failure of a component is only ever manifested as the *absence of a message*. The only way to detect the absence of a message is via a timeout, an ambiguous signal that means either the message will never come or that it merely hasn't come yet. Timeouts are an end-to-end concern[28,48] that must ultimately be managed by the application. Hence, partial failures in distributed systems bubble up the stack and frustrate any attempts at abstraction.

The Old Guard

The modern myth: formally verified distributed components.

If we cannot rely on geniuses to hide the specter of partial failure, the next best hope is to face it head on, armed with tools. Until quite recently, many of us (academics in particular) looked to formal methods such as model checking[16,20,29,39,40,53,54] to assist "mere mortal" programmers in writing distributed code that upholds its guarantees despite pervasive uncertainty in distributed executions. It is not reasonable to exhaustively search the state space of large-scale systems (one cannot, for example, model check Netflix), but the hope is that modularity and composition (the next best tools for conquering complexity) can be brought to bear. If individual distributed components could be formally verified and combined

into systems in a way that preserved their guarantees, then global fault tolerance could be obtained via composition of local fault tolerance.

Unfortunately, this, too, is a pipe dream. Most model checkers require a formal specification; most real-world systems have none (or have not had one since the design phase, many versions ago). Software model checkers and other program-analysis tools require the source code of the system under study. The accessibility of source code is also an increasingly tenuous assumption. Many of the data stores targeted by tools such as Jepsen are closed source; large-scale architectures, while typically built from open-source components, are increasingly *polyglot* (written in a wide variety of languages).

Finally, even if you assume that specifications or source code are available, techniques such as model checking are not a viable strategy for ensuring that applications are fault tolerant because, as discussed in the previous section, in the context of timeouts, fault tolerance itself is an end-to-end property that does not necessarily hold under composition. Even if you are lucky enough to build a system out of individually verified components, it does not follow that the system is fault tolerant—you may have made a critical error in the glue that binds them.

The Vanguard

The emerging ethos: YOLO

Modern distributed systems are simply too large, too heterogeneous, and too dynamic for these classic approaches to software quality to take root. In reaction, practitioners increasingly rely on resiliency techniques based on *testing* and *fault injection*[6,14,19,23,27,35]. These "black box" approaches (which perturb and observe the complete system, rather than its components) are (arguably) better suited for testing an end-to-end property such as fault tolerance. Instead of deriving guarantees from understanding how a system works on the *inside*, testers of the system observe its behavior from the *outside*, building confidence that it functions correctly under stress.

Two giants have recently emerged in this space: Chaos Engineering[6] and Jepsen testing.[24] Chaos Engineering, the practice of actively perturbing production systems to increase overall site resiliency, was pioneered by Netflix,[6] but since then Linkedin,[52] Microsoft,[38] Uber,[47] and PagerDuty[5] have developed Chaos-based infrastructures. Jepsen performs black box testing and fault injection on unmodified distributed data management systems, in search of correctness violations (e.g., counterexamples that show an execution was not linearizable).

Both approaches are pragmatic and empirical. Each builds an understanding of how a system operates under faults by *running the system* and observing its behavior. Both approaches offer a pay-as-you-go method to resiliency: the initial cost of integration is low, and the more experiments that are performed, the higher the confidence that the system under test is robust. Because these approaches represent a straightforward enrichment of existing best practices in testing with well-understood fault injection techniques, they are easy to adopt. Finally, and perhaps most importantly, both approaches have been shown to be effective at identifying bugs.

Unfortunately, both techniques also have a fatal flaw: they are manual processes that require an *extremely* sophisticated operator. Chaos Engineers are a highly specialized subclass of site reliability engineers. To devise a custom fault injection strategy, a Chaos Engineer typically meets with different service teams to build an understanding of the idiosyncrasies of various components and their interactions. The Chaos Engineer then targets those services and interactions that seem likely to have latent fault tolerance weaknesses. Not only is this approach difficult to scale since it must be repeated for every new composition of services, but its critical currency—a mental model of the system under study—is hidden away in a person's brain. These points are reminiscent of a bigger (and more worrying) trend in industry toward reliability priesthoods,[7] complete with icons (dashboards) and rituals (playbooks).

Jepsen is in principle a framework that anyone can use, but to the best of our knowledge all of the reported bugs discovered by Jepsen to date were discovered by its inventor, Kyle Kingsbury, who currently operates a "distributed systems safety research" consultancy.[24] Applying Jepsen to a storage system requires that the superuser carefully read the system documentation, generate workloads, and observe the externally visible behaviors of the system under test. It is then up to the operator to choose—from the massive combinatorial space of "nemeses," including machine crashes and network partitions—those fault schedules that are likely to drive the system into returning incorrect responses.

A human in the loop is the kiss of death for systems that need to keep up with software evolution. Human attention should always be targeted at tasks that computers cannot do! Moreover, the specialists that Chaos and Jepsen testing require are expensive and rare. The remainder of this article shows how geniuses can be abstracted away from the process of failure testing.

We Don't Need Another Hero

Rapidly changing assumptions about our visibility into distributed system internals have made obsolete many if not all of the classic approaches to software quality, while emerging "chaos-based" approaches are fragile and unscalable because of their genius-in-the-loop requirement.

This section presents our vision of automated failure testing by looking at how the same changing environments that hastened the demise of time-tested resiliency techniques can enable new ones. We argue that the best way to automate the experts out of the failure-testing loop is to imitate their best practices in software and show how the emergence of sophisticated observability infrastructure makes this possible.

The Order is Rapidly Fadin'

For large-scale distributed systems, the three fundamental assumptions of traditional approaches to software quality are quickly fading in the rearview mirror. The first to go was the belief that you could rely on experts to solve the hardest problems in the domain. Second was the assumption that a formal specification of the system is available. Finally, any program analysis (broadly defined) that requires that source code is available must be taken off the table. The erosion of these assumptions helps explain the move away from classic academic approaches to resiliency in favor of the black box approaches described earlier.

What hope is there of understanding the behavior of complex systems in this new reality? Luckily, the fact that it is harder than ever to understand distributed systems from the inside has led to the rapid evolution of tools that allow us to understand them from the *outside*. Call-graph logging was first described by Google;[51] similar systems are in use at Twitter,[4] Netflix,[1] and Uber,[50] and the technique has since been standardized.[43] It is reasonable to assume that a modern microservice-based Internet enterprise will already have instrumented its systems to collect call-graph traces. A number of startups that focus on observability have recently emerged.[21,34] Meanwhile, provenance collection techniques for data processing systems [11,22,42] are becoming mature, as are operating system-level provenance tools.[44] Recent work[12,55] has attempted to infer causal and communication structure of distributed computations from raw logs, bringing high-level explanations of outcomes within reach even for uninstrumented systems.

Away from the Experts

Regarding testing distributed systems. Chaos Monkey, like they mention, is awesome, and I also highly recommend getting Kyle to run Jepsen tests. —Commentator on HackerRumor

While this quote is anecdotal, it is hard to imagine a better example of the fundamental unscalability of the current state of the art. A single person cannot possibly keep pace with the explosion of distributed system implementations. If we can take the human out of this critical loop, we must; if we cannot, we should probably throw in the towel.

The first step to understanding how to automate any process is understanding the human component that we would like to abstract away. How do Chaos Engineers and Jepsen superusers apply their unique genius in practice? The remainder of this section describes the three-step recipe common to both approaches.

Step 1: Observe the System in Action

The human element of the Chaos and Jepsen processes begins with principled observation, broadly defined.

A Chaos Engineer will, after studying the external API of services relevant to a given class of interactions, meet with the engineering teams to better understand the details of the implementations of the individual services.[25] To understand the high-level interactions among services, the engineer will then peruse call-graph traces in a trace repository.[3]

A Jepsen superuser typically begins by reviewing the product documentation, both to determine the guarantees that the system should uphold and to learn something about the mechanisms by which it does so. From there, the superuser builds a model of the behavior of the system based on interaction with the system's external API. Since the systems under study are typically data management and storage, these interactions involve generating histories of reads and writes.[31]

The first step to understanding what can go wrong in a distributed system is watching things go right: observing the system in the common case.

Step 2: Build a Mental Model of how the System Tolerates Faults

The common next step in both approaches is the most subtle and subjective. Once there is a mental model of how a distributed system behaves (at least in the common case), how is it

used to help choose the appropriate faults to inject? At this point we are forced to dabble in conjecture: bear with us.

Fault tolerance is redundancy. Given some fixed set of faults, we say that a system is "fault tolerant" exactly if it operates correctly in all executions in which those faults occur. What does it mean to "operate correctly"? Correctness is a system-specific notion, but, broadly speaking, is expressed in terms of properties that are either *maintained* throughout the system's execution (e.g., system invariants or safety properties) or *established* during execution (e.g., liveness properties). Most distributed systems with which we interact, though their executions may be unbounded, nevertheless provide finite, bounded interactions that have *outcomes*. For example, a broadcast protocol may run "forever" in a reactive system, but each broadcast delivered to *all* group members constitutes a successful execution.

By viewing distributed systems in this way, we can revise the definition: a system is fault tolerant if it provides sufficient mechanisms to achieve its successful outcomes despite the given class of faults.

Step 3: Formulate Experiments that Target Weaknesses in the Facade

If we could understand all of the ways in which a system can obtain its good outcomes, we could understand which faults it can tolerate (or which faults it could be sensitive to). We assert that (whether they realize it or not!) the process by which Chaos Engineers and Jepsen superusers determine, on a system-by-system basis, which faults to inject uses precisely this kind of reasoning. A target experiment should exercise a combination of faults that knocks out *all* of the supports for an expected outcome.

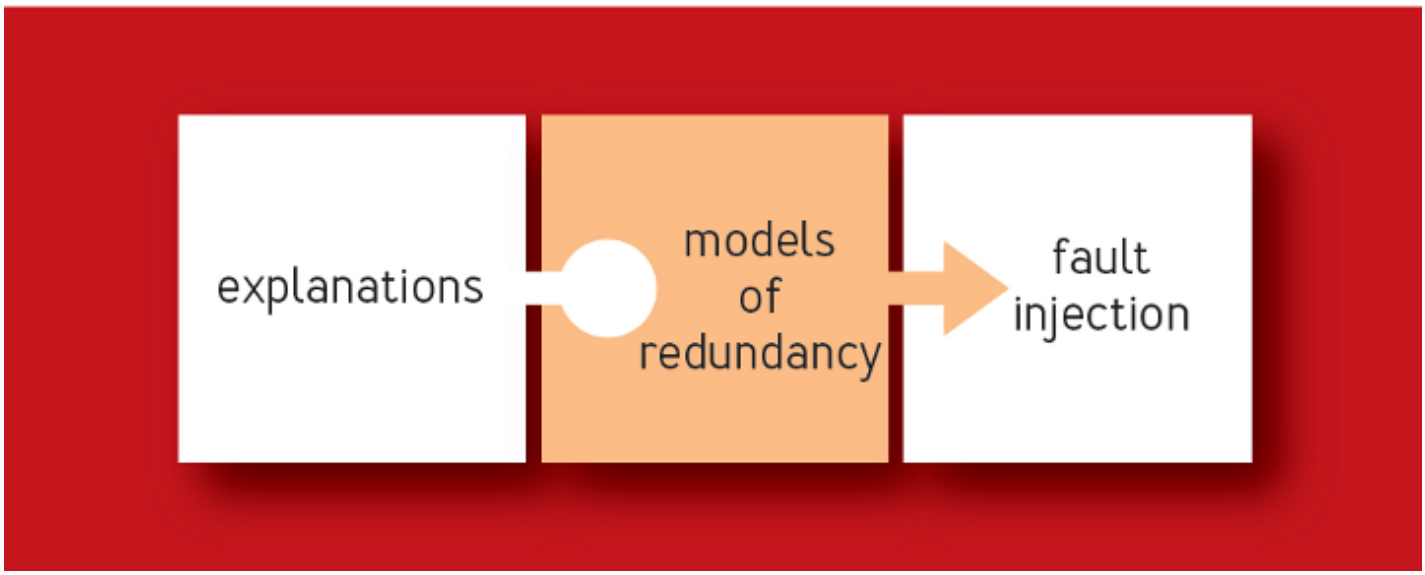
Carrying out the experiments turns out to be the easy part. Fault injection infrastructure, much like observability infrastructure, has evolved rapidly in recent years. In contrast to random, coarse-grained approaches to distributed fault injection such as Chaos Monkey,[23] approaches such as FIT (failure injection testing)[17] and Gremlin[32] allow faults to be injected at the granularity of individual requests with high precision.

Step 4: Profit!

This process can be effectively automated. The emergence of sophisticated tracing tools described earlier makes it easier than ever to build redundancy models even from the

executions of black box systems. The rapid evolution of fault injection infrastructure makes it easier than ever to test fault hypotheses on large-scale systems. Figure 1 illustrates how the automation described in this section fits neatly between existing observability infrastructure and fault injection infrastructure, consuming the former, maintaining a model of system redundancy, and using it to parameterize the latter. Explanations of system outcomes and fault injection infrastructures are already available. In the current state of the art, the puzzle piece that fits them together (*models of redundancy*) is a manual process. LDFI (explained in the next section) shows that automation of this component is possible.

FIGURE 1: **OUR VISION OF AUTOMATED FAILURE TESTING**



A Blast from the Past

In previous work, we introduced a bug-finding tool called LDFI (lineage-driven fault injection).[2] LDFI uses data provenance collected during simulations of distributed executions to build *derivation graphs* for system outcomes. These graphs function much like the models of system redundancy described earlier in the "Away from the Experts" section. LDFI then converts the derivation graphs into a Boolean formula whose satisfying assignments correspond to combinations of faults that invalidate all derivations of the outcome. An experiment targeting those faults will then either expose a bug (i.e., the expected outcome fails to occur) or reveal additional derivations (e.g., after a timeout, the system fails over to a backup) that can be used to enrich the model and constrain future solutions.

At its heart, LDFI reapplies well-understood techniques from data management systems, treating fault tolerance as a materialized view maintenance problem.[2,13] It models a

distributed system as a query, its expected outcomes as query outcomes, and critical facts such as "replica A is up at time t" and "there is connectivity between nodes X and Y during the interval i . . . j" as base facts. It can then ask a how-to query[37]: What changes to base data will cause changes to the derived data in the view? The answers to this query are the faults that could, according to the current model, invalidate the expected outcomes.

The idea seems far-fetched, but the LDFI approach shows a great deal of promise. The initial prototype demonstrated the efficacy of the approach at the level of protocols, identifying bugs in replication, broadcast, and commit protocols.[2,46] Notably, LDFI reproduced a bug in the replication protocol used by the Kafka distributed log[26] that was first (manually) identified by Kingsbury.[30] A later iteration of LDFI is deployed at Netflix,[1] where (much like the illustration in figure 1) it was implemented as a microservice that consumes traces from a call-graph repository service and provides inputs for a fault injection service. Since its deployment, LDFI has identified 11 critical bugs in user-facing applications at Netflix.[1]

Rumors from the Future

The prior research presented in the previous section is only the tip of the iceberg. Much work still needs to be undertaken to realize the vision of fully automated failure testing for distributed systems. This section highlights nascent research that shows promise and identifies new directions that will help realize our vision.

Don't Overthink Fault Injection

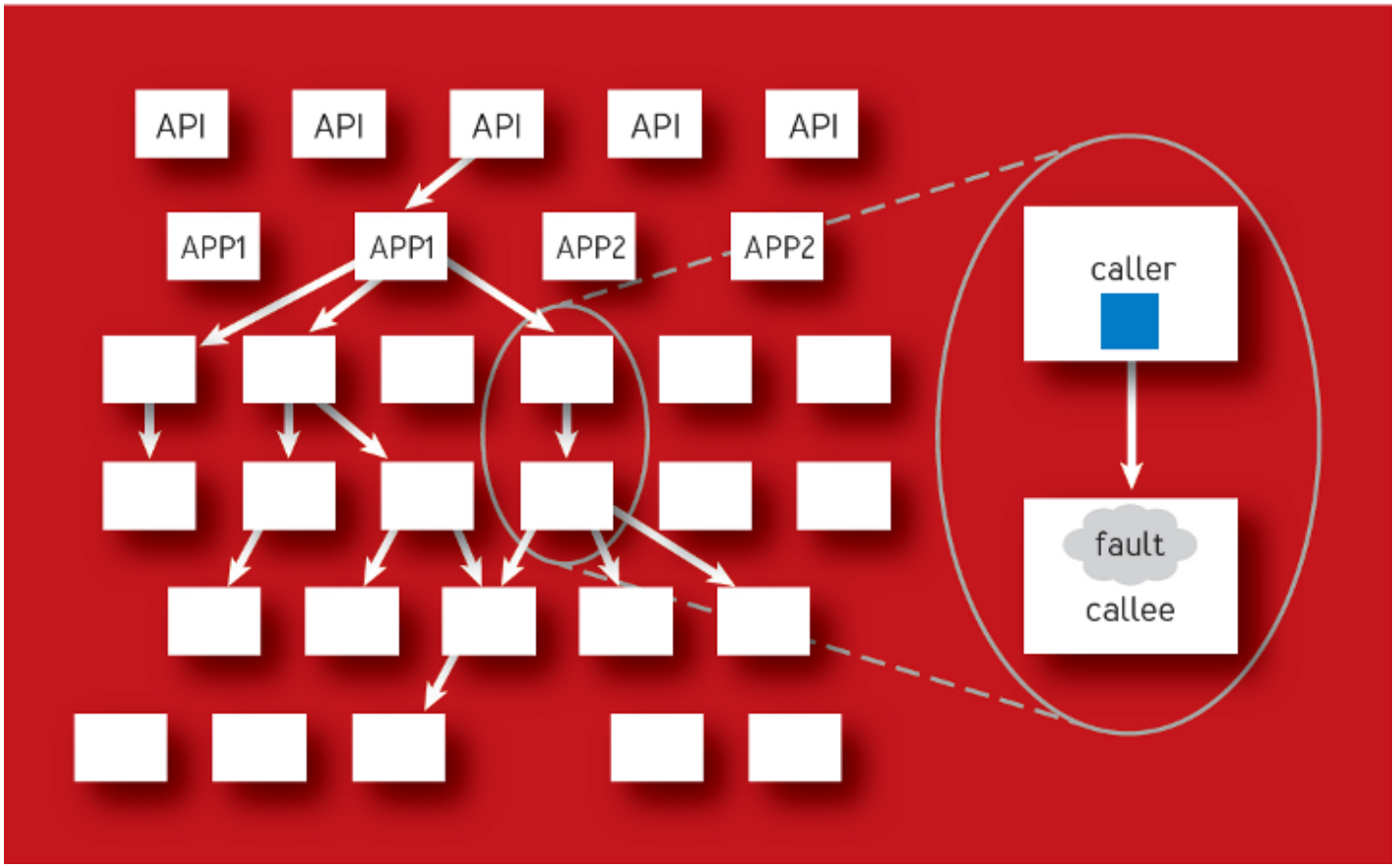
In the context of resiliency testing for distributed systems, attempting to enumerate and faithfully simulate every possible kind of fault is a tempting but distracting path. The problem of understanding all the *causes* of faults is not directly relevant to the target, which is to ensure that code (along with its configuration) intended to detect and mitigate faults performs as expected.

Consider figure 2: the diagram on the left shows a microservice-based architecture; arrows represent calls generated by a client request. The right-hand side zooms in on a pair of interacting services. The shaded box in the caller service represents the fault tolerance logic that is intended to detect and handle faults of the callee. Failure testing targets bugs in this

logic. The fault tolerance logic targeted in this bug search is represented as the shaded box in the caller service, while the injected faults affect the callee.

The common *effect* of all faults, from the perspective of the caller, are explicit error returns, corrupted responses, and (possibly infinite) delay. Of these manifestations, the first two can be adequately tested with unit tests. The last is difficult to test, leading to branches of code that are infrequently executed. If we inject *only* delay, and only at component boundaries, we conjecture that we can address the majority of bugs related to fault tolerance.

FIGURE 2: FAULT INJECTION AND FAULT-TOLERANT CODE



Explanations Everywhere

If we can provide better explanations of system outcomes, we can build better models of redundancy. Unfortunately, a barrier to entry for systems such as LDFI is the unwillingness of software developers and operators to instrument their systems for tracing or provenance collection. Fortunately, operating system–level provenance-collection techniques are mature and can be applied to uninstrumented systems.

Moreover, the container revolution makes simulating distributed executions of black box software within a single hypervisor easier than ever. We are actively exploring the collection of system call–level provenance from unmodified distributed software in order to select a custom-tailored fault injection schedule. Doing so requires extrapolating application-level causal structure from low-level traces, identifying appropriate *cut points* in an observed execution, and finally synchronizing the execution with fault injection actions.

We are also interested in the possibility of inferring high-level explanations from even noisier signals, such as raw logs. This would allow us to relax the assumption that the systems under study have been instrumented to collect execution traces. While this is a difficult problem, work such as the Mystery Machine[12] developed at Facebook shows great promise.

Toward Better Models

The LDFI system represents system redundancy using derivation graphs and treats the task of identifying possible bugs as a materialized-view maintenance problem. LDFI was hence able to exploit well-understood theory and mechanisms from the history of data management systems research. But this is just one of many ways to represent how a system provides alternative computations to achieve its expected outcomes.

A shortcoming of the LDFI approach is its reliance on assumptions of determinism. In particular, it assumes that if it has witnessed a computation that, under a particular contingency (i.e., given certain inputs and in the presence of certain faults), produces a successful outcome, then any future computation under that contingency will produce the same outcome. That is to say, it ignores the uncertainty in timing that is fundamental to distributed systems. A more appropriate way to model system redundancy would be to embrace (rather than abstracting away) this uncertainty.

Distributed systems are probabilistic by nature and are arguably better modeled probabilistically. Future directions of work include the probabilistic representation of system redundancy and an exploration of how this representation can be exploited to guide the search of fault experiments. We encourage the research community to join in exploring alternative internal representations of system redundancy.

Turning the Explanations inside out

Most of the classic work on data provenance in database research has focused on aspects related to human-computer interaction. Explanations of why a query returned a particular result can be used to debug both the query and the initial database—given an unexpected result, what changes could be made to the query or the database to fix it? By contrast, in the class of systems we envision (and for LDFI concretely), explanations are part of the internal language of the reasoner, used to construct models of redundancy in order to drive the search through faults.

Ideally, explanations should play a role in both worlds. After all, when a bug-finding tool such as LDFI identifies a counterexample to a correctness property, the job of the programmers has only just begun—now they must undertake the onerous job of distributed debugging. Tooling around debugging has not kept up with the explosive pace of distributed systems development. We continue to use tools that were designed for a single site, a uniform memory, and a single clock. While we are not certain what an ideal distributed debugger should look like, we are quite certain that it does not look like GDB (GNU Project debugger).[36] The derivation graphs used by LDFI show how provenance can also serve a role in debugging by providing a concise, visual explanation of how the system reached a bad state.

This line of research can be pushed further. To understand the root causes of a bug in LDFI, a human operator must review the provenance graphs of the good and bad executions and then examine the ways in which they differ. Intuitively, if you could abstractly *subtract* the (incomplete by assumption) explanations of the bad outcomes from the explanations of the good outcomes,[10] then the root cause of the discrepancy would be likely to be near the "frontier" of the difference.

Conclusion

A sea change is occurring in the techniques used to determine whether distributed systems are fault tolerant. The emergence of fault injection approaches such as Chaos Engineering and Jepsen is a reaction to the erosion of the availability of expert programmers, formal specifications, and uniform source code. For all of their promise, these new approaches are crippled by their reliance on superusers who decide which faults to inject.

To address this critical shortcoming, we propose a way of modeling and ultimately automating the process carried out by these superusers. The enabling technologies for this vision are the rapidly improving observability and fault injection infrastructures that are becoming commonplace in the industry. While LDFI provides constructive proof that this approach is

possible and profitable, it is only the beginning. Much work remains to be done in targeting faults at a finer grain, constructing more accurate models of system redundancy, and providing better explanations to end users of *exactly what went wrong* when bugs are identified. The distributed systems research community is invited to join in exploring this new and promising domain.

Related Articles

Fault Injection in Production
Making the case for resilience testing
– John Allspaw, Etsy
http://queue.acm.org/detail.cfm?id=2353017

The Verification of a Distributed System
A practitioner's guide to increasing confidence in system correctness
– Caitie McCaffrey
http://queue.acm.org/detail.cfm?id=2889274

Injecting Errors for Fun and Profit
Error-detection and correction features are only as good as our ability to test them.
– Steve Chessin, Oracle
http://queue.acm.org/detail.cfm?id=1839574

References

1. Alvaro, P., Andrus, K., Basiri, A., Hochstein, L., Rosenthal, C., Sanden, C. 2016. Automating failure testing research at Internet scale. In *Proceedings of the 7th ACM Symposium on Cloud Computing:* 17–28.

2. Alvaro, P., Rosen, J., Hellerstein, J. M. 2015. Lineage-driven fault injection. In *Proceedings of the ACM SIGMOD International Conference on Management of Data:* 331–346.

3. Andrus, K. 2016. Personal communication.

4. Aniszczyk, C. 2012. Distributed systems tracing with Zipkin. Twitter Engineering; https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin.

5. Barth, D. 2014. Inject failure to make your systems more reliable. DevOps.com; http://devops.com/2014/06/03/inject-failure/.

6. Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C. 2016. Chaos Engineering. *IEEE Software* 33(3): 35–41.

7. Beyer, B., Jones, C., Petoff, J., Murphy, N. R. 2016. *Site Reliability Engineering*. O'Reilly.

8. Birrell, A. D., Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2(1): 39–59.

9. Chandra, T. D., Hadzilacos, V., Toueg, S. 1996. The weakest failure detector for solving consensus. *Journal of the Association for Computing Machinery* 43(4): 685–722.

10. Chen, A., Wu, Y., Haeberlen, A., Zhou, W., Loo, B. T. 2016. The good, the bad, and the differences: better network diagnostics with differential provenance. In *Proceedings of the ACM SIGCOMM Conference:* 115–128.

11. Chothia, Z., Liagouris, J., McSherry, F., Roscoe, T. 2016. Explaining outputs in modern data analytics. *Proceedings of the VLDB Endowment* 9(12): 1137–1148.

12. Chow, M., Meisner, D., Flinn, J., Peek, D., Wenisch, T. F. 2014. 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:* 217–231.

13. Cui, Y., Widom, J., Wiener, J. L. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems* 25(2): 179–227.

14. Dawson, S., Jahanian, F., Mitton, T. 1996. ORCHESTRA: A Fault Injection Environment for Distributed Systems. In *Proceedings of the 26th International Symposium on Fault-tolerant Computing.*

15. Fischer, M. J., Lynch, N. A., Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery* 32(2): 374–382; https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf.

16.Fisman, D., Kupferman, O., Lustig, Y. 2008. On verifying fault tolerance of distributed protocols. In *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science* 4963: 315–331. Springer Verlag.

17. Andrus, K., Gopalani, N., Schmaus, B. 2014. FIT: failure injection testing. Netflix Technology Blog; http://techblog.netflix.com/2014/10/fit-failure-injection-testing.html.

18. Gray, J. 1985. Why do computers stop and what can be done about it? Tandem Technical Report 85.7; http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf.

19. Gunawi, H. S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J. M., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Sen, K., Borthakur, D. 2011. FATE and DESTINI: a framework for cloud recovery testing. In *Proceedings of the 8th Usenix Conference on Networked Systems Design and Implementation:* 238-252; http://db.cs.berkeley.edu/papers/nsdi11-fate-destini.pdf.

20. Holzmann, G. 2003. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional.

21. Honeycomb. 2016; https://honeycomb.io/.

22. Interlandi, M., Shah, K., Tetali, S. D., Gulzar, M. A., Yoo, S., Kim, M., Millstein, T., Condie, T. 2015. Titian: data provenance support in Spark. In *Proceedings of the VLDB Endowment* 9(3): 216-227.

23. Izrailevsky, Y., Tseitlin, A. 2011. The Netflix Simian Army. Netflix Technology Blog; http://techblog.netflix.com/2011/07/netflix-simian-army.html.

24. Jepsen. 2016. Distributed systems safety research; http://jepsen.io/.

25. Jones, N. 2016. Personal communication.

26. Kafka 0.8.0. 2013. Apache; https://kafka.apache.org/08/documentation.html.

27. Kanawati, G. A., Kanawati, N. A., Abraham, J. A. 1995. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers* 44(2): 248-260.

28. Kendall, S. C., Waldo, J., Wollrath, A., Wyant, G. 1994. A note on distributed computing. Technical Report. Sun Microsystems Laboratories.

29. Killian, C. E., Anderson, J. W., Jhala, R., Vahdat, A. 2007. Life, death, and the critical transition: finding liveness bugs in systems code. In *Networked System Design and Implementation*: 243-256.

30. Kingsbury, K. 2013. Call me maybe: Kafka; http://aphyr.com/posts/293-call-me-maybe-kafka.

31. Kingsbury, K. 2016. Personal communication.

32. Lafeldt, M. 2017. The discipline of Chaos Engineering. Gremlin Inc.; https://blog.gremlininc.com/the-discipline-of-chaos-engineering-e39d2383c459.

33. Lampson, B. W. 1980. Atomic transactions. In *Distributed Systems—Architecture and Implementation, An Advanced Course:* 246-265;

https://link.springer.com/chapter/10.1007%2F3-540-10571-9_11.

34. LightStep. 2016; http://lightstep.com/.

35. Marinescu, P. D., Candea, G. 2009. LFI: a practical and general library-level fault injector. In *IEEE/IFIP International Conference on Dependable Systems and Networks*.

36. Matloff, N., Salzman, P. J. 2008. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press.

37. Meliou, A., Suciu, D. 2012. Tiresias: the database oracle for how-to queries. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*: 337-348.

38. Microsoft Azure Documentation. 2016. Introduction to the fault analysis service; https://azure.microsoft.com/en-us/documentation/articles/service-fabric-testability-overview/.

39. Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., Dill, D. L. 2002. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review.* In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* 36(SI): 75-88.

40. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A. Neamtiu, I. 2008. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th Usenix Conference on Operating Systems Design and Implementation:* 267-280.

41. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M. 2014. Use of formal methods at Amazon Web Services. Technical Report; http://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf.

42. Olston, C., Reed, B. 2011. Inspector Gadget: a framework for custom monitoring and debugging of distributed data flows. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data:* 1221-1224.

43. OpenTracing. 2016; http://opentracing.io/.

44. Pasquier, T. F. J. -M., Singh, J., Eyers, D. M., Bacon, J. 2015. CamFlow: managed data-sharing for cloud services; https://arxiv.org/pdf/1506.04391.pdf.

45. Patterson, D. A., Gibson, G., Katz, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data:* 109-116; http://web.mit.edu/6.033/2015/wwwdocs/papers/Patterson88.pdf.

46. Ramasubramanian, K., Dahlgren, K., Karim, A., Maiya, S., Borland, S., Alvaro, P. 2017. Growing a protocol. In *9th Usenix Workshop on Hot Topics in Cloud Computing*.

47. Reinhold, E. 2016. Rewriting Uber engineering: the opportunities microservices provide. Uber Engineering; https://eng.uber.com/building-tincup/.

48. Saltzer, J. H., Reed, D. P., Clark, D. D. 1984. End-to-end arguments in system design. *ACM Transactions on Computing Systems* 2(4): 277–288.

49. Sandberg, R. 1986. The Sun network file system: design, implementation and experience. Technical report, Sun Microsystems. In *Proceedings of the Summer 1986 Usenix Technical Conference and Exhibition.*

50. Shkuro, Y. 2017. Jaeger: Uber's distributed tracing system. Uber Engineering; https://uber.github.io/jaeger/.

51. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C. 2010. Dapper, a large-scale distributed systems tracing infrastructure. Technical report. Research at Google; https://research.google.com/pubs/pub36356.html.

52. Shenoy, A. 2016. A deep dive into Simoorg: our open source failure induction framework. Linkedin Engineering; https://engineering.linkedin.com/blog/2016/03/deep-dive-Simoorg-open-source-failure-induction-framework.

53. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L. 2009. MODIST: Transparent model checking of unmodifed distributed systems. In *Proceedings of the 6th Usenix Symposium on Networked Systems Design and Implementation:* 213–228.

54. Yu, Y., Manolios, P., Lamport, L. 1999. Model checking TLA+ specifcations. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 99):* 54–66.

55. Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D., Stumm, M. 2014. Lprof: a non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th Usenix Conference on Operating Systems Design and Implementation:* 629–644.

Peter Alvaro is an assistant professor of computer science at the University of California Santa Cruz, where he leads the Disorderly Labs research group (disorderlylabs.github.io). His research focuses on using datacentric languages and analysis techniques to build and reason

about data-intensive distributed systems in order to make them scalable, predictable, and robust to the failures and nondeterminism endemic to large-scale distribution. He earned his Ph.D. at UC Berkeley, where he studied with Joseph M. Hellerstein. He is a recipient of the NSF CAREER award.

Severine Tymon is a technical writer who has written documentation for both internal and external users of enterprise and open-source software. She has worked at such industry leaders as Pivotal, Microsoft, CNET, VMware, and Oracle. Special interests include researching and documenting fault tolerance in distributed systems. Tymon earned her bachelor's degree in literature and creative writing at UC Berkeley.

Originally published in Queue vol. 15, no. 5—
Comment on this article in the ACM Digital Library

More related articles:

Phil Vachon - **The Keys to the Kingdom**
An unlucky fat-fingering precipitated the current crisis: The client had accidentally deleted the private key needed to sign new firmware updates. They had some exciting new features to ship, along with the usual host of reliability improvements. Their customers were growing impatient, but my client had to stall when asked for a release date. How could they come up with a meaningful date? They had lost the ability to sign a new firmware release.

Pat Helland, Simon Weaver, Ed Harris - **Too Big NOT to Fail**
Web-scale infrastructure implies LOTS of servers working together, often tens or hundreds of

thousands of servers all working toward the same goal. How can the complexity of these environments be managed? How can commonality and simplicity be introduced?

Steve Chessin – **Injecting Errors for Fun and Profit**

It is an unfortunate fact of life that anything with moving parts eventually wears out and malfunctions, and electronic circuitry is no exception. In this case, of course, the moving parts are electrons. In addition to the wear-out mechanisms of electromigration (the moving electrons gradually push the metal atoms out of position, causing wires to thin, thus increasing their resistance and eventually producing open circuits) and dendritic growth (the voltage difference between adjacent wires causes the displaced metal atoms to migrate toward each other, just as magnets will attract each other, eventually causing shorts), electronic circuits are also vulnerable to background radiation.

Michael W. Shapiro – **Self-Healing in Modern Operating Systems**

Driving the stretch of Route 101 that connects San Francisco to Menlo Park each day, billboard faces smilingly reassure me that all is well in computerdom in 2004. Networks and servers, they tell me, can self-defend, self-diagnose, self-heal, and even have enough computing power left over from all this introspection to perform their owner-assigned tasks.