# Growing a protocol

Kamala Ramasubramanian[1], Kathryn Dahlgren[1], Asha Karim[1], Sanjana Maiya[1], Sarah Borland[1],
Boaz Leskes[2], Peter Alvaro[1]

[1]University of California, Santa Cruz
[2]Elastic
{kramasub, kmdahlgr, akarim, smaiya, sborland, palvaro}@ucsc.edu
boaz@elastic.co

## Abstract

Verification is often regarded as a one-time procedure undertaken after a protocol is specified but before it is implemented. However, in practice, protocols continually evolve with the addition of new capabilities and performance optimizations. Existing verification tools are ill-suited to "tracking" protocol evolution and programmers are too busy (or too lazy?) to simultaneously co-evolve specifications manually. This means that the correctness guarantees determined at verification time can erode as protocols evolve. Existing software quality techniques such as regression testing and root cause analysis, which naturally support system evolution, are poorly suited to reasoning about *fault tolerance* properties of a distributed system because these properties require a search of the execution schedule rather than merely replaying inputs. This paper advocates that our community should explore the intersection of testing and verification to better ensure quality for distributed software and presents our experience evolving a data replication protocol at Elastic using a novel bug-finding technology called Lineage Driven Fault Injection (LDFI) as evidence.

## 1 Introduction

Common distributed systems wisdom warns us *never to reinvent*. If we have a problem requiring consensus, we use Paxos [38] (or Raft [47]); if we need strong consistency data replication for availability, we use Primary/Backup [40] or Chain Replication [56]. To disseminate updates, we use reliable broadcast [42]. Best practices dictate that we invariably choose a well-understood (and, ideally, formally verified) protocol as the basis of our implementation.

Because the protocols used to solve these problems are mature, it might appear that protocol design is mostly a thing of the past: modern systems designers can merely take mechanisms "off the shelf" and enjoy the guarantees of hardened subsystems while constructing otherwise novel applications.

Any practitioner, however, will quickly identify this as a fallacy. Even initial protocol implementations tend to differ significantly from their specification. Furthermore, over the lifetime of a system, protocol details undergo a series of optimizations in response to particular use cases. Since such optimizations can range from the fussy (e.g., tweaking timeout parameters) to the fundamental (e.g., bypassing protocol steps based on assumptions about the common case), it can be challenging to know which implementation changes are tantamount to changes in the specification (which would in principle then need to be reverified). Such a circumstance places implementors in the bad position of deriving false confidence from assertions that their implementation is "essentially Primary/Backup".

Software engineering best practices provide us with a variety of tools for ensuring program correctness over the course of a development lifecycle. For example, regression testing techniques ensure future optimizations do not re-introduce bugs previously encountered in earlier stages of system development. When dormant bugs manifest in later system versions, root cause analysis techniques allow us to replay "bad inputs" over the commit history until we identify the version in which the bug was introduced.

Unfortunately, all of these techniques associate aberrant *behaviors* (i.e. bugs) with the *inputs* that trigger them. A regression test ensures a bug triggered by a given input is never re-introduced by making the replay of the input part of the regression suite. Root cause analysis identifies the first version in which a bug appears, by replaying the particular input that triggered it at all previous commits.

Fault tolerance properties of distributed systems, by contrast, assert the system computes a correct outcome even in the face of a predefined class of faults, such as machine crashes and network partitions. Consequently, the classic software quality techniques described above are useless. Subtle changes to protocols can fundamentally affect fault tolerance characteristics; seemingly innocuous modifications may trigger incorrect behaviors. Notably, an input known to trigger a bug in a particular version of the protocol is *not* guaranteed to trigger *the same bug* in a different version. As a result, regression testing, as we currently employ it, is fundamentally too weak to prevent fault tolerance regression bugs. Root cause analysis is similarly inadequate, because a set of faults triggering bugs in later versions may fail to do so in an earlier version.

In this paper, we argue that tool support for implementing and evolving fault-tolerant distributed systems needs to be rethought. We advocate exploration of the (sparse) middle ground between existing testing techniques practically inadequate for addressing fault tolerance concerns and traditional verification techniques ill-suited to the continual evolution of real-world implementations. We describe our experience using a novel bug-finding methodology called Lineage-Driven Fault Injection [11] to test a new replication protocol developed at Elasticsearch [49]. We show how the lightweight verification approach makes it possible to apply software engineering best practices such as regression testing and root cause analysis in the context of fault tolerance properties and identify desiderata for future tools.

## 2 Motivation

Distributed systems prove an unwieldy challenge to the mature quality methodologies we typically apply to evolving software. In particular, issues arise because fault tolerance properties are sensitive to a space of faults as opposed to specific inputs. In traditional methods of testing, bugs are characterized by inputs, whereas in distributed systems they are tied to the execution schedule. It is this disparity that necessitates use of a different tool for testing and verification for fault tolerance.

Consider, for instance, a distributed system that relies on a leader election module. Version 1 of this module implements bully leader election [26], choosing the node with the *minimum* ID as leader, and Version 2 chooses the node with the *maximum* ID. It's not difficult to convince ourselves that these are essentially the same protocol, and as a result, it might not occur to us to re-verify. Imagine now that there is a bug in the downstream logic: if the leader crashes, it fails to uphold its invariants.

If we first encounter this bug in Version 1, following best practices, we might write a *regression test* that injects a fault into node **x** in all future tests. However, that input is not sufficient to trigger the aberrant behavior in Version 2. Though we have the same bug in Version 2, crashing **x** may not trigger it. Instead, crashing node **y** will. For us to discover **y** as a trigger, we would need to back-port the protocol changes to the specification and re-verify.

Conversely, if we first encounter the bug in Version 2, we might perform *root cause analysis* and work backwards through the commit history, replaying the failure of **y** in earlier versions. Yet we would still fail to detect the bug because, in Version 1, a failure in **x**, not **y**, triggers the behavior. Again, we find that we must re-verify for every commit if we hope to discover the bug.

We are left wanting something that works like verification, but feels like testing. We need to perform a principled search of the space of execution schedules while retaining the efficiency, tool support, and integration provided by existing testing practices. This search has to be run on every commit, but an exhaustive search of the space of possible combinations of faults is intractable. There is a need to prune the space of potential faults we must explore for testing and verification, but a dearth of tools available to do so.

## 3 Background

This paper is based on insights from a summer internship at Elastic, a distributed data store vendor whose products focus on real-time search and analytics of documents [27]. At the time, their engineering team had deployed a data replication protocol based on Primary/Backup. Any Primary/Backup protocol needs a way to sync a stale copy of the data with the current primary. The Elasticsearch protocol uses a method based on file syncing to do so. Since file copying is inherently slow, Elastic was looking for a faster protocol that can work by synchronizing individual operations and avoiding the overhead of copying large files. The new protocol would work without pauses in writes and allow indexing to multiple documents concurrently. Since this was a new algorithm, Elastic was looking for ways to formally verify it. However, most verification tools require specifications and it is not reasonable to think that every time a programmer comes up with an optimization, they implement it in code and add it to the specification. Therefore, they favored an efficient, lightweight tool designed for easy use and incorporating strong failure scenario exploration guarantees.

Elastic engaged our research team because they wanted a technique that strikes a balance between formal verification and testing—in particular, the strong correctness guarantees of the former and the agility of the latter. Lineage Driven Fault Injection (LDFI) is an analysis and fault selection framework that harnesses concepts from logic programming and database theory to construct a representation of the underlying system model and derive explanations for behaviors under different fault scenarios [11]. The approach builds a model based on a good system execution and explores only fault scenarios capable of forcing the system into a bad state. This optimization greatly reduces the space of examined fault scenarios, making it tractable to perform a search of the fault space each time the underlying implementation changes (even trivially).

We implemented a sequence of versions of the replication protocol and used LDFI to incrementally verify them as a part of continuous integration. The following section describes our experiences using LDFI to examine the impact of the modifications on overall correctness.

## 4 Modeling the replication protocol

The core Elasticsearch data replication protocol is a variation of primary backup. All client requests are routed to the primary and are only acknowledged after they have been replicated by the primary on all replicas. While building the system, we defined incomplete versions of the protocol starting with the core functionality, the last version being as close to the real system as possible. Each version, as a result of being incomplete, had historical bugs. Discovering these issues that were not caught by conventional software engineering techniques gave us confidence that the approach is effective.

Since the Elasticsearch API guarantees focus around a single document, we modeled a single document with concurrent accesses, rather than multiple independent documents. For simplicity, we focused on an cluster with one primary and two replicas. To simplify the evaluation process, the specification also allows the existence of a master oracle omniscient with respect to the state of all other processes in the system. The master oracle abstracts away the running of some correct consensus protocol internally on a group of servers.

### 4.1 Catching Bugs Early

There are many instances in the software development cycle for a bug to be introduced, the first of which is when a protocol specification is converted to an implementation. During our case study, we found a bug which manifested precisely from such a translation scenario. As illustrated in Figure 1, after sending two concurrent writes to two different nodes in the system, LDFI tested a scenario in which one of the writes is replicated successfully while the second write is replicated on only one of the replicas. Then, before the second write replicates on the other node, the primary fails over. Subsequently, the node on which the latest write request has not been replicated becomes the new primary. The two replicas are now (and will forever remain) inconsistent.

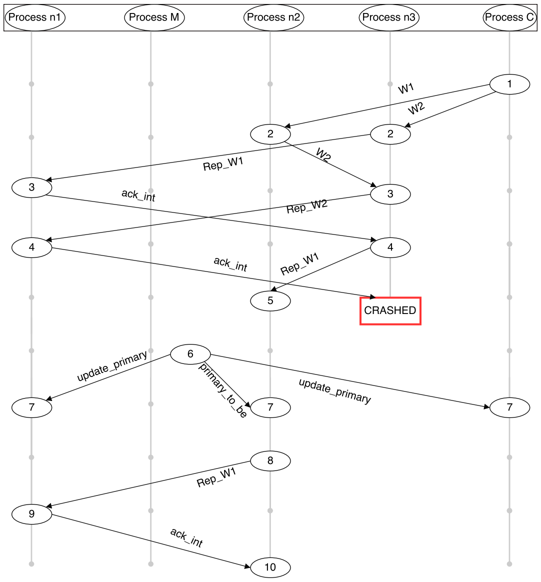

Figure 1: *Concurrent-writes bug. Process M represents the mast oracle; Process C represents the client; and all Process nN processes represent active replicas.*

Discovering the bug requires primary failure after launching successful writes to only a subset of backup replicas. Furthermore, a replica from the unlucky subset must become the new primary. The main difficulty in catching this bug using techniques such as test-driven development or regression testing is the manual derivation of relevant test cases. LDFI offers a better alternative by generating such scenarios automatically. The technique analyses the flow of data throughout the system for a simulated correct execution and iteratively examines the protocol's responses to different message drop/process crash combinations.

### 4.2 Dormant Bugs

When we discover a bug, we would like to go back in history to determine the version at which the bug was introduced. This is because a bug can lie dormant for a long time before it is discovered. As an example, after discovering the bug with concurrent writes, we were able to reproduce the bug in the case in which there was only a single write. Figure 2 represents this exact scenario. As can be seen, the two bugs are similar, but do not manifest from the same fault scenarios. This reinforces the claim from our motivating example that techniques such as root cause analysis as they are generally deployed would not be effective in reasoning about the fault tolerance properties of distributed systems.

In this particular case, in a system supporting concurrent writes, we would have witnessed the same interactions as the single write scenario with appropriate in-

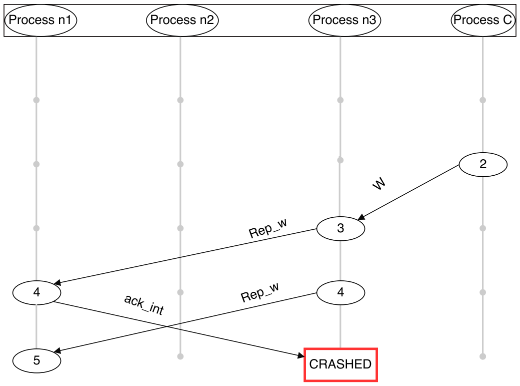

Figure 2: *Concurrent-writes bug occurs in the single-write scenario as well*

put data. This brings into sharp focus the fact that the input data we start with matters in finding interesting bugs.We discuss the problem of simultaneously searching the space of faults and inputs in Section 5.

## 4.3 Optimizations

Once a protocol implementation exists, practitioners naturally optimize for performance or carry out functionality extensions. However, some optimizations may change the specification and without further verification, we cannot (or at least shouldn't) offer statements regarding correctness.

### 4.3.1 Sequence Number Optimization

A seemingly minor optimization can result in a serious fault tolerance bug. In Elasticsearch, the primary locally chooses monotonically increasing sequence numbers to enforce ordering on concurrent requests. Sequence numbers were introduced to prevent newer data from being overwritten. To avoid extra processing, the following rule was applied: *If the sequence number associated with a write request has been seen before, drop the payload but acknowledge the request.*

Now consider a scenario in which the primary fails over after sending write requests from a client to a subset of the backup replicas. Suppose further that a replica ignorant of the write takes over as the primary and receives a new write request. Since sequence numbers are *locally* determined by the primary, it may pick the same sequence number as the incomplete write. It will then send the write to all the active replicas. However, some replicas may drop the write in adherence to the above optimization. Figure 3 demonstrates one instance of such an execution. Fortunately, LDFI quickly and automatically discovers such a scenario by using the initial successful execution to test fault scenarios that may cause failures.

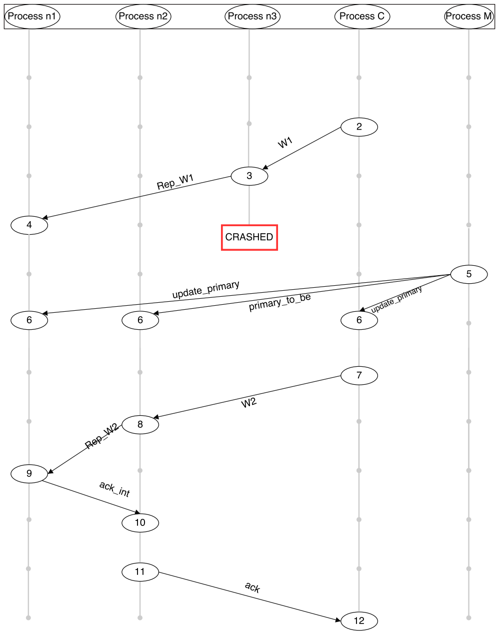

Figure 3: *Optimizing for sequence numbers*

The above represents just one scenario in which verification can catch bugs in optimizations. Optimization carries the risk of introducing entirely new bugs capable of breaking the end-to-end properties of the system, which is best handled by verification-based tools.

### 4.3.2 Checkpoint Optimization

When a new node is promoted as primary, a re-sync is necessary to ensure that all the active replicas in the system are consistent with the new primary. In the initial model, all writes were replayed to the replicas. However, this is extremely expensive and inefficient as only operations that weren't acknowledged to the client need to be replayed. Therefore, we model a checkpoint optimization using local and global checkpoints to ensure the entire history of acknowledged messages is not resent to replicas upon the election of a new primary. Each replica maintains a local checkpoint while a global checkpoint is the minimum of all local checkpoints. A newly elected primary only sends update messages to replicas possessing a sequence number greater than the global checkpoint. This variation of the protocol introduces a fair amount of complexity, but produces no counterexamples when run against LDFI.

Simplicity of an optimization is not a consideration in determining if the correctness guarantees of a system have been violated. In this section, we demonstrated how a seemingly simple optimization breaks system guarantees while another more intricate one doesn't.

# 5 Past and Future Work

Our experience at Elastic suggests approaches like LDFI are a step towards improving the state of the art in distributed software quality. In this section, we place our work in context between the past developments upon which it builds and the work that we hope will follow.

The protocol described in Section 4 is a variant of Primary/Backup [5, 40, 55], a well-understood data replication technique that (when correctly implemented!) ensures single-copy consistency. As we argue in Section 1, protocols such as Primary/Backup are continually reinterpreted and extended in practice and developers (ever-optimistic by nature) derive false confidence from the abstract connections to "correct" protocols.

**Concurrency bugs:** On one side of the spectrum, mature verification techniques such as model checking [24, 33, 45, 58, 59]—particularly the software model checkers capable of verifying real implementations [14, 37, 44] — are ideally suited for reasoning about *concurrency bugs* triggered by nondeterministic scheduling orders. Unfortunately, verifying fault tolerance properties of distributed systems with state space exploration techniques like model checking is challenging due to the combinatorial explosion of possible faults [29, 30, 39].

Recent work on semantic-aware software model checkers (e.g. SAMC [39]) is particularly encouraging. These tools require encoding domain knowledge about any independence and symmetry characteristic to the problem to dramatically reduce the state space under consideration. Such a process supports the efficient exploration of the system execution behaviors dependent upon complex patterns of faults and orderings.

An ideal tool solution would combine the best features of LDFI (which automatically builds models of domain knowledge, but ignores concurrency and asynchrony) with state-of-the-art combined approaches such as SAMC, since we know from Fischer et al. [23] that some of the most fundamental difficulties of distributed systems exist at the intersection of partial failure and asynchrony! LDFI's roots in data-centric programming languages suggest a unique approach to tackling concurrency bugs. The CALM Theorem [7, 12, 32], which asserts *monotonic* programs invariably produce deterministic outcomes for all message delivery orders, provides an insight into how event orderings either necessitate or avoid race conditions at runtime. *We are developing a prototype system that combines the Lineage-Driven approach (utilizing explanations of* what went right *to reason about what could go wrong) and CALM analysis (using static analysis to prove commutativity of message processing logic) to simultaneously prune the space of faults and re-orderings.*

**System Models:** On the other side of the spectrum, fault injection frameworks [1, 2, 22, 25, 29, 30, 52] are maturing. Approaches such as LDFI are complementary to fault injection techniques and can be used to automatically drive such classes of debugging efforts as substantiated by Alvaro et al. [6]. LDFI is just one example of a more general technique: build models of system redundancy from observability infrastructure (e.g. tracing systems) and use those models to prune the space of faults to inject. *Given how probabilistic models are arguably more appropriate to the domain of distributed systems, we anticipate future work on LDFI embracing rather than masking the inherent uncertainties in timing endemic to distributed executions.*

**Input Generation:** In this paper, we assume the inputs to the system are given *a priori* and focus computational resources on fault selection. However, in practice, it can be tricky to discover the inputs required to trigger a bug. A variety of approaches to input generation and test generation [16, 19, 41, 50] are available. While it is tempting to argue that these techniques are complementary to our approach, the reality is more nuanced. In practice, some fault tolerance bugs in distributed systems are triggered only by specific interleavings of inputs and fault events; Zave's counterexamples [61] to the correctness invariants for Chord [54] provide a compelling witness. *We are pursuing work that co-optimizes the search through faults and inputs.*

**Debugging tools:** When a testing or verification tool identifies a possible bug, the process of *debugging* has only just begun. Much like the quality assurance techniques discussed in Section 2, classic software debugging approaches, as referenced throughout the paper, are ill-fitted to distributed systems. Currently, distributed debugging tool support is in its infancy, so a great many directions are possible. Our experience using LDFI at Elastic suggests the provision of high-level *explanations* of how a system achieves (or fails to achieve) good outcomes are a good starting point for taming the complexity of distributed debugging. Provenance [15, 21, 28, 36, 43, 62] is a well-established model in the database and systems literature for providing explanations of outcomes. *Using provenance to reason about distributed executions, however, is a young research area capable of radical growth in tandem with future improvements in observability infrastructure support [3, 4, 13, 18, 48, 51].*

# 6 Conclusion

Existing bug detection and root cause analysis tools are inadequate for assessing the correctness of distributed protocols. The paper describes our experience seeking a middle ground between formal verification and software testing techniques while developing a novel distributed protocol intended for a real-world, production environment. Given our success, we are optimistic that LDFI is a step in the right direction. However, to be clear, we do not believe in a one-size-fits-all solution. Our experience confirms our intuitions that the future of fault tolerant software development is unlikely to come in the form of a single verification methodology. Rather, we see a future in which tool support for distributed software implementation, evolution, and debugging is improved in a variety of directions. The state of the art is so desperately poor that is should be easy for the research community to make an impact!

**Acknowledgements**

We would like to thank the entire Elasticsearch team and in particular Yannick Welsch and Jason Tedor for their invaluable role in making this paper possible. This work is supported by the National Science Foundation under Grant No. 1652368.

## 7 Discussion

**Open Issues:** Translating protocol designs into Dedalus was, ultimately and unfortunately, a bottleneck. Developing minimally weakened variations of LDFI that shift away from the specification requirement and toward more flexible input formats, such as system execution traces or call-graphs, is an active area of future work that increase general appeal of the technique.

**When Does the Whole Idea Fall Apart?** A number of failure classes exist beyond the scope of the presented LDFI approach. For example, LDFI cannot yet handle complex event interleaving patterns reminiscent of Zave's Chord counterexamples [61], as highlighted in the future work section. Additionally, Byzantine failures are still far beyond the capabilities of the current technique.

**Feedback Solicitation:** In this paper, we identify the need for new methods to optimally harness current software quality best practices for debugging the fault tolerance properties of distributed systems. We are particularly interested in rebuttals against any of our core beliefs, especially:

- Classical software quality techniques such as regression testing and root cause analysis do not extend to distributed systems in their current form.
- LDFI serves as a bridge between verification and testing, as demonstrated by its successful real-world application.

Additionally, the paper demonstrates that classical debugging techniques can be effectively applied to distributed systems with the right intermediary formulations. What other tools should we be building? What potential impact could the LDFI approach have on such tools?

**Type of Discussion and Controversial Points:** Apart from the discussions generated from our assertions above, comparison between techniques such as LDFI and products from the ever-evolving field of model checking would be anticipated discussion topics. As highlighted in the related work, some existing research seeks to expand the power of model checkers for distributed systems applicability. Are techniques in the intersection of testing and verification valuable if such efforts succeed? Furthermore, will the future landscape of distributed software debuggers essentially manifest as a variation of a one-size-fits-all solution or, as we believe, a rich toolset addressing particular classes of debugging needs? We look forward to debating these visions of the future.

## References

[1] The Netflix Simian Army. http://techblog.netflix.com/2011/07/netflix-simian-army.html, 2011.

[2] Nemesis: Disruptive Testing. `https://www.scribd.com/document/318375955/Yahoo-Nemesis`, 2015.

[3] Jaeger. `https://godoc.org/github.com/uber/jaeger-client-go`, 2016.

[4] The OpenTracing Project. `http://opentracing.io/`, 2016.

[5] Alsberg, P. A., and Day, J. D. A Principle for Resilient Sharing of Distributed Resources. ICSE '76.

[6] Alvaro, P., Andrus, K., Sanden, C., Rosenthal, C., Basiri, A., and Hochstein, L. Automating failure testing research at internet scale. In *SoCC* (2016), ACM, pp. 17–28.

[7] Alvaro, P., Conway, N., Hellerstein, J. M., and Marczak, W. R. Consistency Analysis in Bloom: a CALM and Collected Approach. CIDR'12.

[8] Alvaro, P., Hutchinson, A., Conway, N., Marczak, W. R., and Hellerstein, J. M. BloomUnit: Declarative Testing for Distributed Programs. DBTest '12.

[9] Alvaro, P., Hutchinson, A., Conway, N., Marczak, W. R., and Hellerstein, J. M. Bloomunit: Declarative testing for distributed programs. In *Proceedings of the Fifth International Workshop on Testing Database Systems* (2012), ACM, p. 1.

[10] Alvaro, P., Marczak, W. R., Conway, N., Hellerstein, J. M., Maier, D., and Sears, R. Dedalus: Datalog in Time and Space. Datalog'10.

[11] Alvaro, P., Rosen, J., and Hellerstein, J. M. Lineage-driven fault injection. SIGMOD, ACM.

[12] Ameloot, T. J., Neven, F., and Van den Bussche, J. Relational Transducers for Declarative Networking. PODS'12.

[13] Aniszczyk, C. Distributed Systems Tracing with Zipkin. `https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin`, June 2012.

[14] Ball, T., Levin, V., and Rajamani, S. K. A Decade of Software Model Checking with SLAM. *Commun. ACM* (2011).

[15] Buneman, P., Khanna, S., and Tan, W.-c. Why and Where: A Characterization of Data Provenance. ICDT'01.

[16] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. OSDI'08.

[17] CHENEY, J., CHITICARIU, L., AND TAN, W.-C. Provenance in Databases: Why, How, and Where. *Found. Trends databases* (April 2009).

[18] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Oct. 2014).

[19] CLAESSEN, K., AND HUGHES, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP '00.

[20] CLARKE, E., BIERE, A., RAIMI, R., AND ZHU, Y. Bounded Model Checking Using Satisfiability Solving. *Form. Methods Syst. Des.* (July 2001).

[21] CUI, Y., WIDOM, J., AND WIENER, J. L. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* (June 2000).

[22] DAWSON, S., JAHANIAN, F., AND MITTON, T. ORCHESTRA: A Fault Injection Environment for Distributed Systems. Tech. rep., FTCS, 1996.

[23] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* (April 1985).

[24] FISMAN, D., KUPFERMAN, O., AND LUSTIG, Y. On verifying fault tolerance of distributed protocols. In *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963 of *LNCS*. Springer Berlin Heidelberg, 2008.

[25] FIT : Failure Injection Testing. `http://techblog.netflix.com/2014/10/fit-failure-injection-testing.html`, 2014.

[26] GARCIA-MOLINA, H. Elections in a distributed computing system. *IEEE Trans. Comput.* (January 1982).

[27] GORMLEY, C., AND TONG, Z. *Elasticsearch:A definitive guide*.

[28] GREEN, T. J., KARVOUNARAKIS, G., AND TANNEN, V. Provenance semirings. PODS '07.

[29] GUNAWI, H. S., DO, T., HELLERSTEIN, J. M., STOICA, I., BORTHAKUR, D., AND ROBBINS, J. Failure as a service (FaaS): A cloud service for large-scale, online failure drills. Tech. rep., EECS Department, University of California, Berkeley, 2011.

[30] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. FATE and DESTINI: A Framework for Cloud Recovery Testing. NSDI'11.

[31] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (October 2015), ACM.

[32] HELLERSTEIN, J. M. The Declarative Imperative: Experiences and conjectures in distributed logic. *SIGMOD Record* (2010).

[33] HOLZMANN, G. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[34] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. USENIX ATC'10.

[35] JACKSON, D. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[36] KARVOUNARAKIS, G., IVES, Z. G., AND TANNEN, V. Querying data provenance. SIGMOD '10.

[37] KILLIAN, C. E., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. NSDI'07.

[38] LAMPORT, L. The Part-time Parliament. *ACM Transactions on Computer Systems* (May 1998).

[39] LEESATAPORNWONGSA, T., HAO, M., JOSHI, P., LUKMAN, J. F., AND GUNAWI, H. S. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*

[40] LIN, W., YANG, M., ZHANG, L., AND ZHOU, L. PacificA: Replication in Log-Based Distributed Storage Systems. Tech. rep., 2008.

[41] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of unix utilities. *Commun. ACM 33* (1990).

[42] MULLENDER, S., Ed. *Distributed Systems*, second ed. Addison-Wesley, 1993.

[43] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. ATEC '06.

[44] MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.* (2002).

[45] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI'08.

[46] ONGARO, D. Runway: A New Tool for Distributed Systems Design. *USENIX ;login* (Fall 2016).

[47] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. USENIX ATC'14.

[48] PASQUIER, T. F. J., SINGH, J., EYERS, D. M., AND BACON, J. CamFlow: Managed Data-sharing for Cloud Services. *CoRR* (2015).

[49] RAMASUBRAMNAIAN, K., AND LESKES, B. Using molly to model and test data replication in elasticseach, 2 2017.

[50] SEN, K., AND AGHA, G. Automated Systematic Testing of Open Distributed Programs. In *Fundamental Approaches to Software Engineering*, L. Baresi and R. Heckel, Eds., vol. 3922 of *LNCS*. 2006.

[51] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Tech. rep., Google, Inc., 2010.

[52] A Deep Dive into Simoorg: Our Open Source Failure Induction Framework, 2016.

[53] SONG, Y. J., JUNQUEIRA, F., AND REED, B. BFT for the Skeptics. http://www.sigops.org/sosp/sosp09/slides/song-slides-sosp09wip.pdf, 2009. SOSP'09 WIP Talk.

[54] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2001), SIGCOMM '01.

[55] STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Trans. Softw. Eng.* (May 1979).

[56] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. OSDI'04.

[57] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. NSDI'09.

[58] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. NSDI'09.

[59] YU, Y., MANOLIOS, P., AND LAMPORT, L. Model checking tla+ specifications. CHARME '99.

[60] ZAMFIR, C., AND CANDEA, G. Execution Synthesis: A Technique for Automated Software Debugging. EuroSys '10.

[61] ZAVE, P. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev. 42*.

[62] ZHOU, W., SHERR, M., TAO, T., LI, X., LOO, B. T., AND MAO, Y. Efficient querying and maintenance of network provenance at internet-scale. SIGMOD '10.