The Verification of a Distributed System

A Practitioner's Guide to Increasing Confidence in System Correctness

Caitie McCaffrey

Distributed Systems Engineer

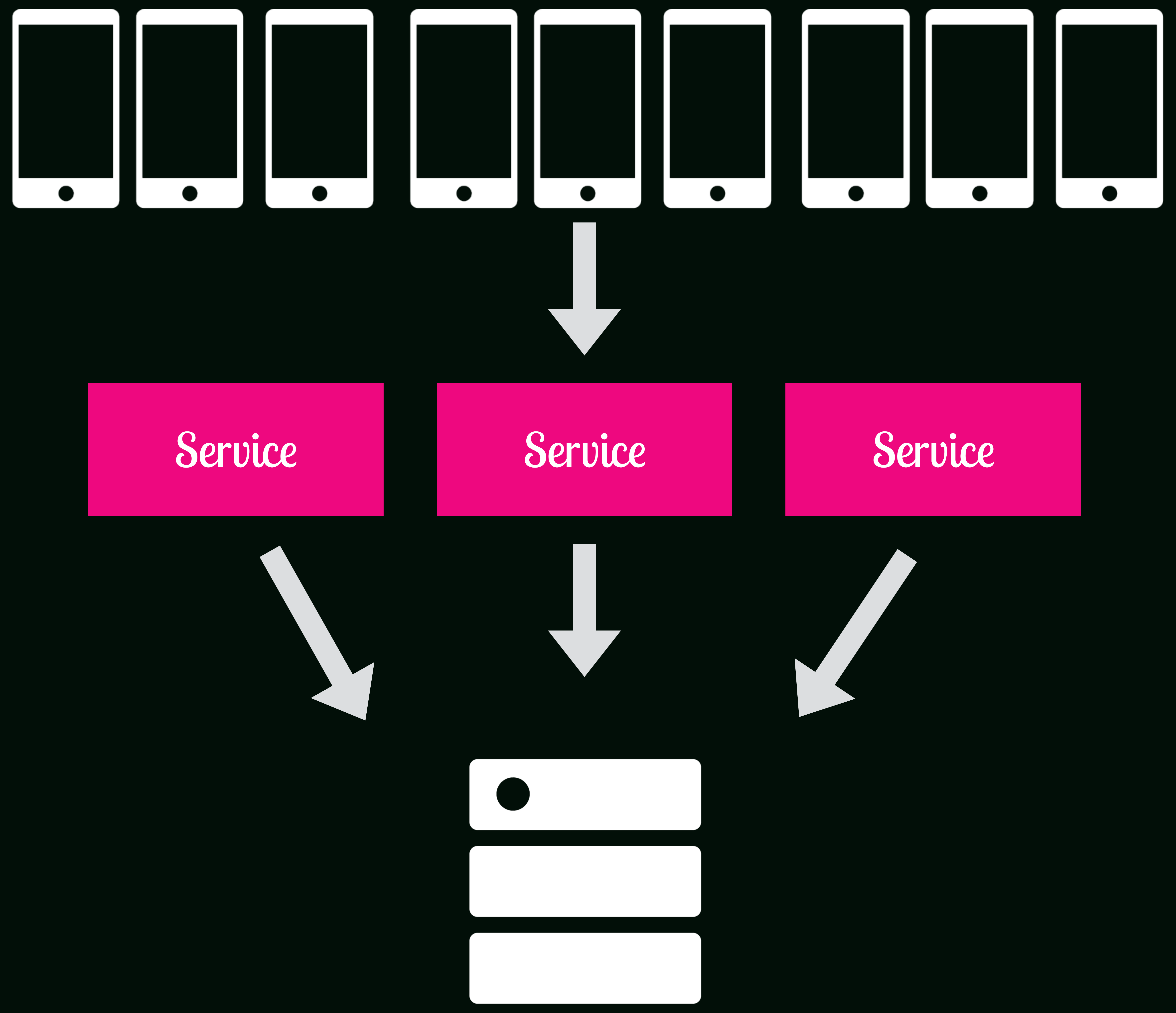
@caitie

caitiem.com

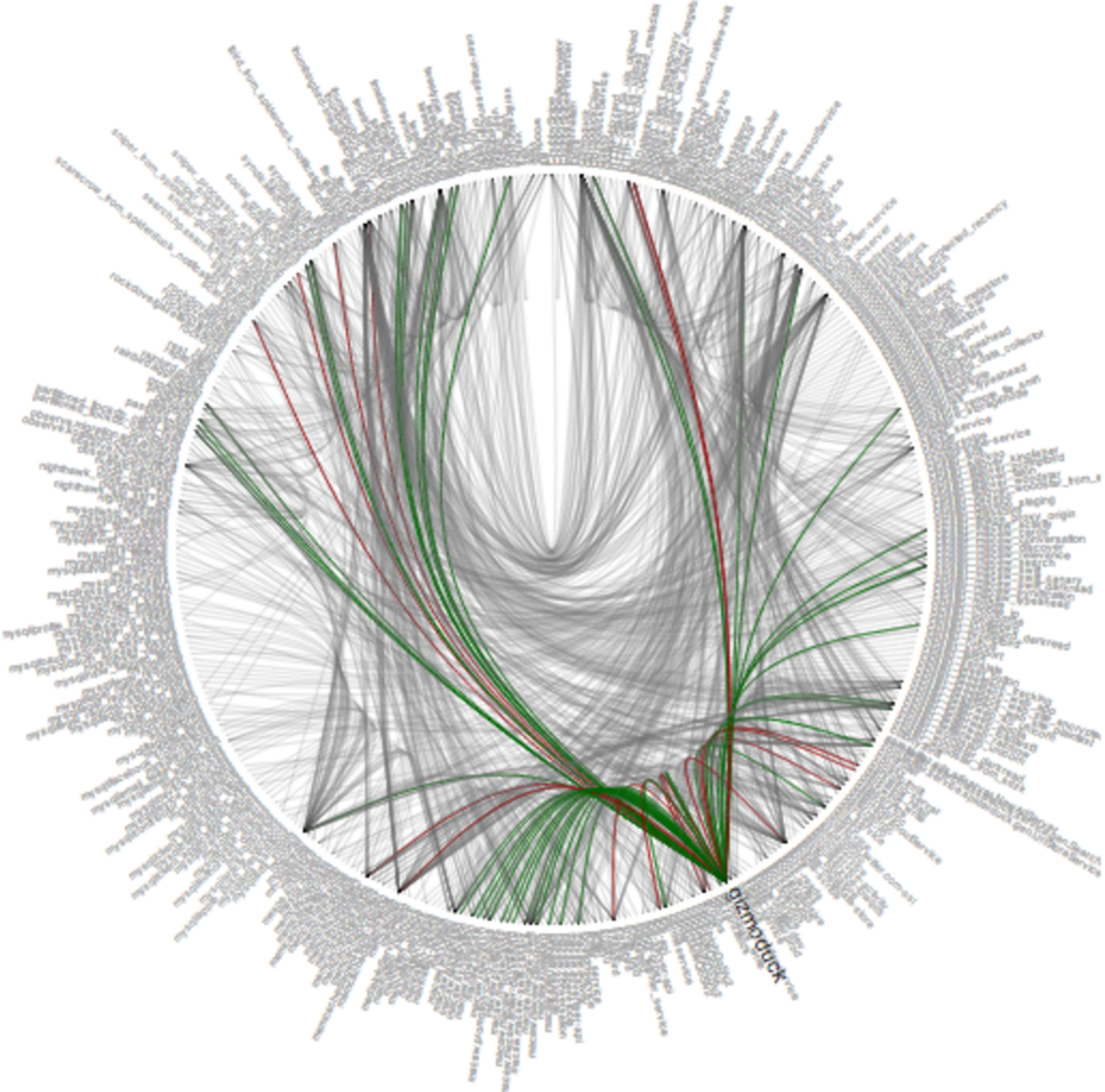
"A Distributed System is one in which the failure of a computer you didn't even know existed can render your own computer unusable"

LESLIE LAMPORT

We Are All Building Distributed Systems



Twitter Services



What the hell have you built.

- Did you just pick things at random?
- Why is Redis talking to MongoDB?
- Why do you even *use* MongoDB?

Goddamnit

Nevermind FREE

Overview

Formal Verification

Provably Correct Systems

Testing in the Wild

Increase Confidence in System Correctness

Research

A New Hope

References

CaitieM20/TheVerificationO

GitHub, Inc. [US] https://github.com/CaitieM20/TheVerificationOfDistributedSystem

The Verification of a Distributed System

Accompanying Repository for The Verification of a Distributed System Talk to be given at GOTO C

Abstract

Distributed Systems are difficult to build and test for two main reasons: partial failure & asynchron
distributed systems must be addressed to create a correct system, and often times the resulting s
degree of complexity. Because of this complexity, testing and verifying these systems is critically i
will discuss strategies for proving a system is correct, like formal methods, and less strenuous me
help increase our confidence that our systems are doing the right thing.

References

- The Verification of a Distributed System
- Specifying Systems
- Use of Formal Methods at Amazon Web Services
- Simple Testing Can Prevent Most Critical Failures
- Property Based Testing
  - Haskell: Quick Check
  - Erlang: Quick Check
  - Other Quick Check Implementations
  - ScalaCheck
  - 29 GIFs only ScalaCheck Witches will Understand

Safety & Liveness

Formal Verification

Formal Verification

Formal Specifications

Written description of what a system is supposed to do

TLA+ ★ Coq

TLA+

"Its a good idea to understand a system before building it, so its a good idea to write a specification of a system before implementing it"

Leslie Lamport, Specifying Systems

★ TLA+ Hour Clock Specification ★

```
———————— MODULE HourClock ————————
EXTENDS Naturals
VARIABLE hr
HCini == hr \in (1 .. 12)
HCnxt == hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC == HCini /\ [][HCnxt] _hr
——————————————————————————————————
THEOREM HC => []HCini
==================================
```

Leslie Lamport, Specifying Systems

TLA+

Use of Formal Methods at Amazon Web Services

Use of Formal Methods at Amazon Web Services

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Amazon.com

29th September, 2014

Since 2011, engineers at Amazon Web Services (AWS) have been using formal specification and model checking to help solve difficult design problems in critical systems. This paper describes our motivation and experience, what has worked well in our problem domain, and what has not. When discussing personal experiences we refer to authors by their initials.

At AWS we strive to build services that are simple for customers to use. That external simplicity is built on a hidden substrate of complex distributed systems. Such complex internals are required to achieve high availability while running on cost-efficient infrastructure, and also to cope with relentless rapid business growth. As an example of this growth; in 2006 we launched S3, our Simple Storage Service. In the 6 years after launch, S3 grew to store 1 trillion objects [1]. Less than a year later it had grown to 2 trillion objects, and was regularly handling 1.1 million requests per second [2].

S3 is just one of tens of AWS services that store and process data that our customers have entrusted to us. To safeguard that data, the core of each service relies on fault-tolerant distributed algorithms for replication, consistency, concurrency control, auto-scaling, load balancing, and other coordination tasks. There are many such algorithms in the literature, but combining them into a cohesive system is a major challenge, as the algorithms must usually be modified in order to interact properly in a real-world system. In addition, we have found it necessary to invent algorithms of our own. We work hard t[illegible]oid unnecessary complexity, but the essential complexity of the task remains high.

High complexity increases the probability of human error in design, code, and operations. Err[illegible] core of the system could cause loss or corruption of data, or violate other interface contract[illegible]on which our customers depend. So, before launching such a service, we need to reach extremely high confidence

★ TLA+

"Formal Methods Have Been a Big Success"

S3 & 10+ Core Pieces of Infrastructure Verified

- - - - - - - - - -

2 Serious Bugs Found

- - - - - - - - - -

Increased Confidence to make Optimizations

Applying TLA+ to some of our more complex systems

System	Components	Line count (excl. comments)	Benefit
S3	Fault-tolerant low-level network algorithm	804 PlusCal	Found 2 bugs. Found further bugs in proposed optimizations.
	Background redistribution of data	645 PlusCal	Found 1 bug, and found a bug in the first proposed fix.
DynamoDB	Replication & group-membership system	939 TLA+	Found 3 bugs, some requiring traces of 35 steps
EBS	Volume management	102 PlusCal	Found 3 bugs.
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence. Failed to find a liveness bug as we did not check liveness.
	Fault tolerant replication and reconfiguration algorithm	318 TLA+	Found 1 bug. Verified an aggressive optimization.

Use of Formal Methods at Amazon Web Services

“Formal methods deal with models of systems, not the systems themselves”

Use of Formal Methods at Amazon Web Services

Distributed Systems Testing in the Wild

Distributed Systems Testing in the Wild

"Seems Pretty Legit"

Unit Tests

Testing of Individual Software Components or Modules

TYPES ARE NOT TESTING

A Short Counter Example

```scala
/*
 * Add two numbers together
 */
def Add (x: Int, y: Int):Int = {
  x * y
}

Add(4, 3)
```

Scala


TCP DOESN'T CARE ABOUT
YOUR TYPE SYSTEM

Integration Tests

Testing of integrated modules to verify combined functionality

Simple Testing Can Prevent Most Critical Failures

Simple Testing Can Prevent Most Critical Failures

An Analysis of Production Failures in Distributed Data-intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao,
Yongle Zhang, Pranay U. Jain, Michael Stumm
University of Toronto

Abstract

Large, production quality distributed systems still fail periodically, and do so sometimes catastrophically, where most or all users experience an outage or data loss. We present the result of a comprehensive study investigating 198 randomly selected, user-reported failures that occurred on Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis, with the goal of understanding how one or multiple faults eventually evolve into a user-visible failure. We found that from a testing point of view, almost all failures require only 3 or fewer nodes to reproduce, which is good news considering that these services typically run on a very large number of nodes. However, multiple inputs are needed to trigger the failures with the order between them being important. Finally, we found the error logs of these systems typically contain sufficient data on both the errors and the input events that triggered the failure, enabling the diagnose and the reproduction of the production failures.

We found the majority of catastrophic failures could easily have been prevented by performing simple testing on error handling code – the last line of defense – even without an understanding of the software design. We extracted three simple rules from the bugs that have lead to some of the catastrophic failures, and developed a static checker, Aspirator, capable of locating these bugs. Over 30% of the catastrophic failures would have been pre-

raises the questions of *why these systems still experience failures* and *what can be done to increase their resiliency*. To help answer these questions, we studied 198 randomly sampled, user-reported failures of five data-intensive distributed systems that were designed to tolerate component failures and are widely used in production environments. The specific systems we considered were Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis.

Our goal is to better understand the specific failure manifestation sequences that occurred in these systems in order to identify opportunities for improving their availability and resiliency. Specifically, we want to better understand how one or multiple errors[1] evolve into component failures and how some of them eventually evolve into service-wide catastrophic failures. Individual elements of the failure sequence have previously been studied in isolation, including root causes categorizations [33, 52, 50, 56], different types of causes including misconfiguraitons [43, 66, 49], bugs [12, 41, 42, 51] hardware faults [62], and the failure symptoms [33, 56], and many of these studies have made significant impact in that they led to tools capable of identifying many bugs (e.g., [16, 39]). However, the entire manifestation sequence connecting them is far less well-understood.

For each failure considered, we carefully studied the failure report, the discussion between users and developers, the logs and the code, and we manually reproduced

Three nodes or less can reproduce 98% of failures

Testing error handling code could have prevented 58% of catastrophic failures

Simple Testing can Prevent Most Critical Failures

35% of Catastrophic Failures

Error Handling Code is simply empty or only contains a Log statement

Error Handler aborts cluster on an overly general exception

Error Handler contains comments like FIXME or TODO

Simple Testing can Prevent Most Critical Failures

Property Based Testing

QuickCheck

Haskell
&
Erlang

ScalaCheck

Scala
&
Java

Languages with Quick Check Ports:

C, C++, Clojure, Common Lisp, Elm, F#, C#, Go, JavaScript, Node.js, Objective-C, OCaml, Perl, Prolog, PHP, Python, R, Ruby, Rust, Scheme, Smalltalk, StandardML, Swift

ScalaCheck Examples

```scala
import org.scalacheck._

val smallInteger = Gen.choose(0,100)
val propSmallInteger = Prop.forAll(smallInteger) { n =>
  n >= 0 && n <= 100
}
```

```scala
import org.scalacheck._

val propReverseList = forAll { l:List[String] => l.reverse.reverse == l }
```

Fault Injection

Introducing faults into the system under test

"Without explicitly forcing a system to fail, it is unreasonable to have any confidence it will operate correctly in failure modes"

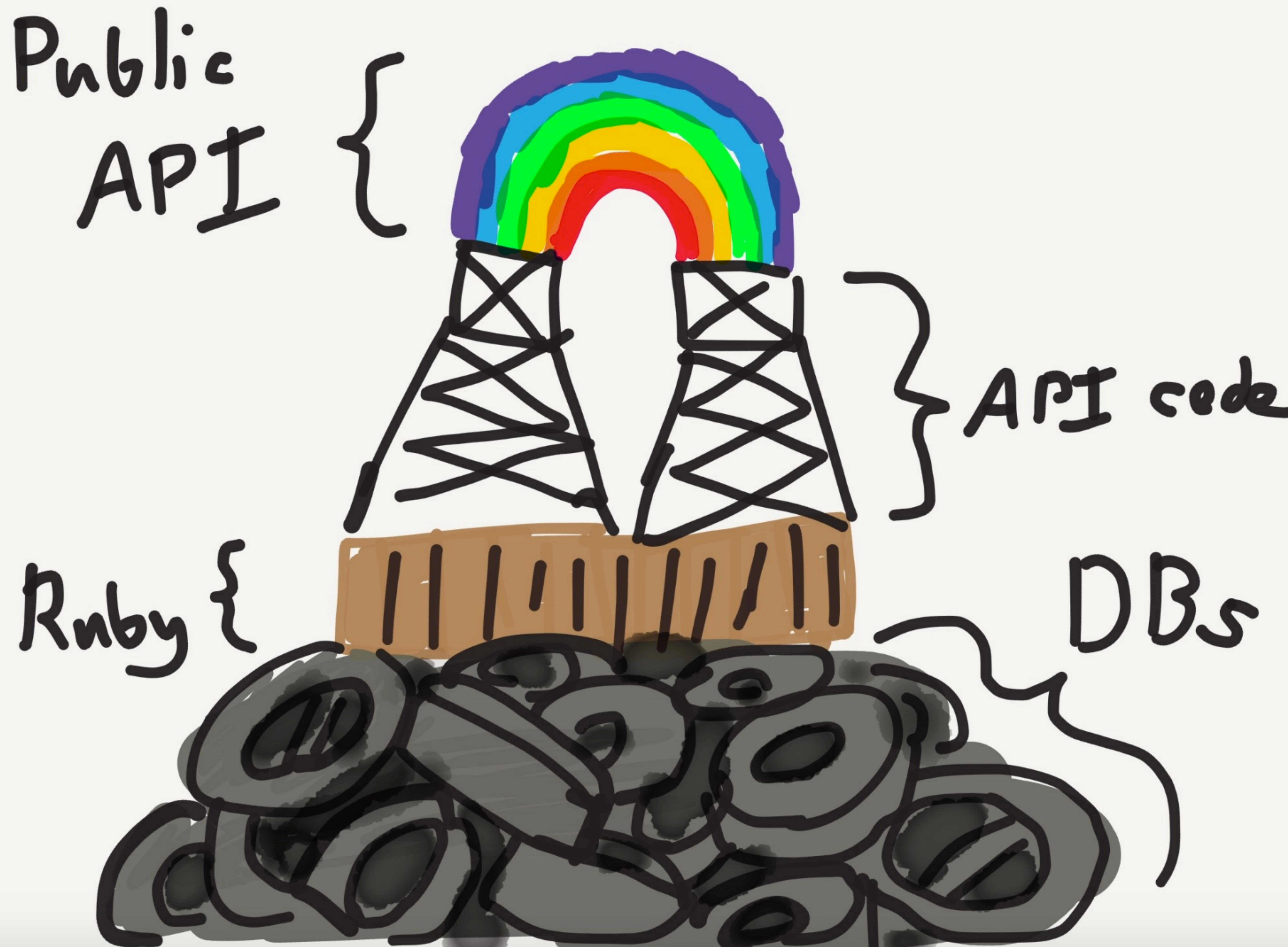
-The Verification of a Distributed System

Netflix Simian Army

- **Chaos Monkey:** kills instances
- **Latency Monkey:** artificial latency induced
- **Chaos Gorilla:** simulates outage of entire availability zone.

JEPSEN

Fault Injection Tool that simulates network partitions in the system under test



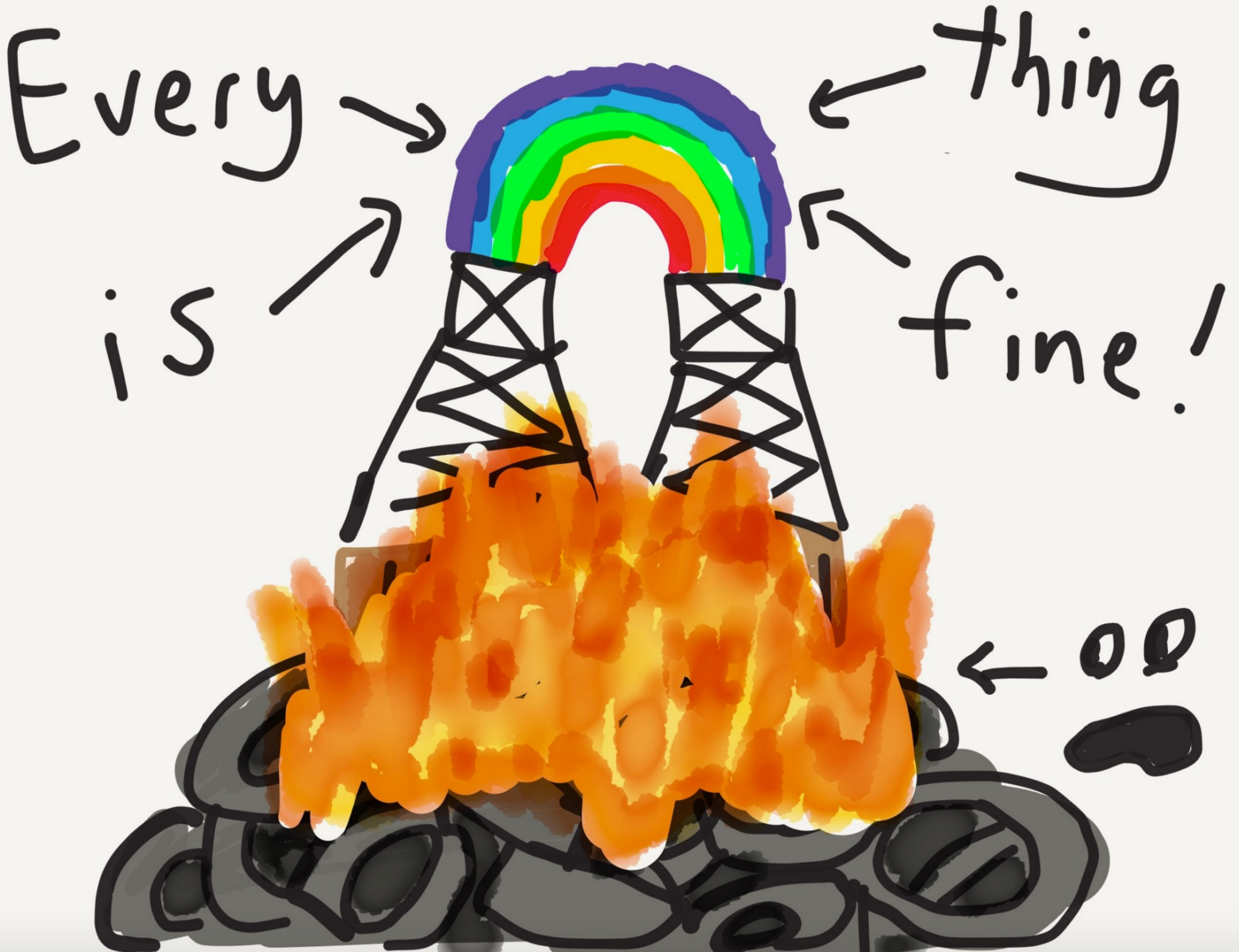
credit: @aphyr

JEPSEN

Fault Injection Tool that simulates network partitions in the system under test

Kyle

credit: @aphyr



CAUTION: Passing Tests Does Not Ensure Correctness

GAME DAYS

Breaking your services on purpose

How to Run a GameDay

1. Notify Engineering Teams that Failure is Coming
2. Induce Failures
3. Monitor Systems Under Test
4. Observing Only Team Monitors Recovery Processes & Systems, Files Bugs
5. Prioritize Bugs & Get Buy-In Across Teams

Game Day at Stripe

"During a recent game day, we tested failing over a Redis cluster by **running kill -9** on its primary node, and ended up **losing all data** in the cluster"

Game Day Exercises at Stripe: Learning from `kill -9`

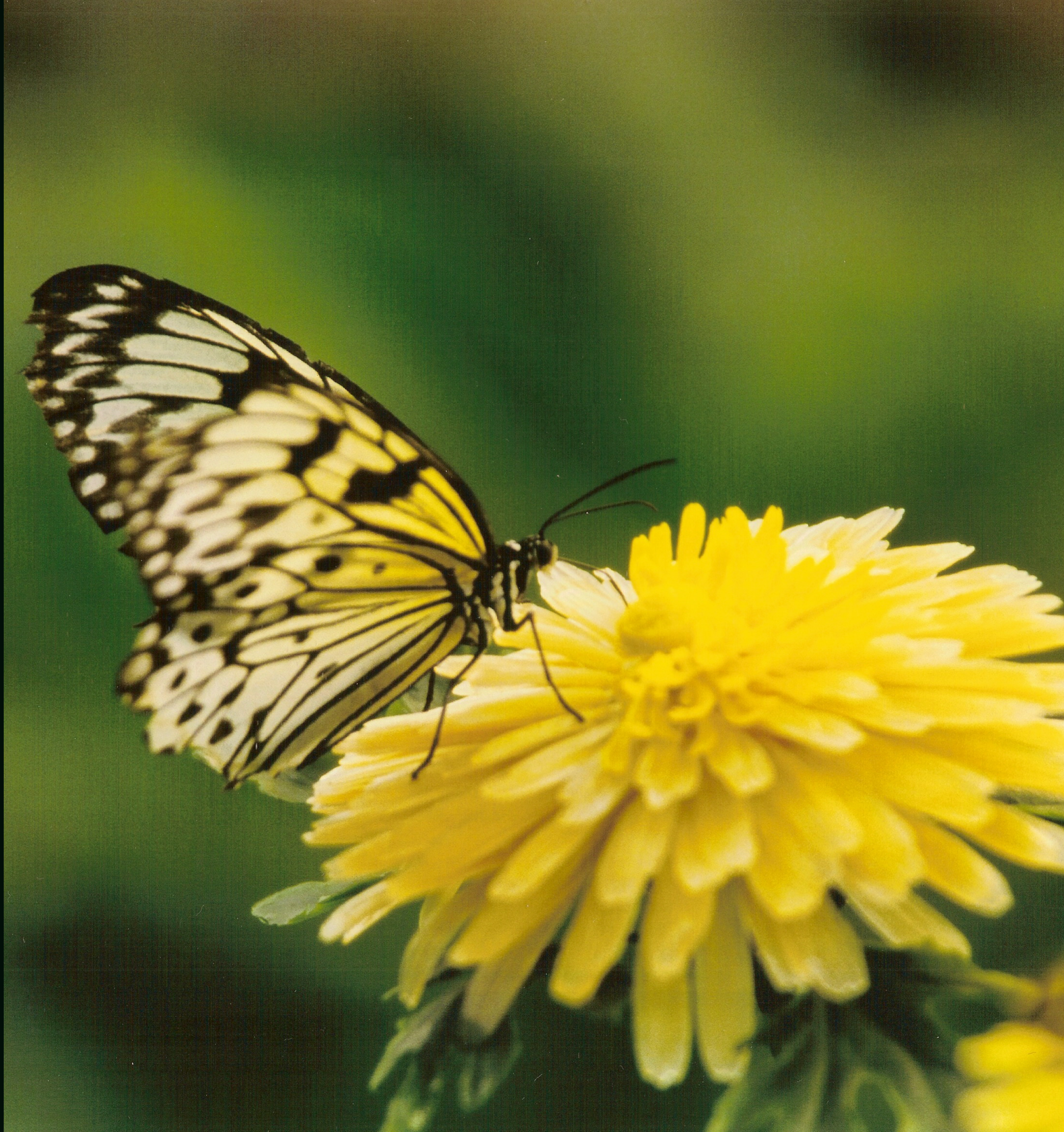
Some thoughts on

TESTING IN PRODUCTION

MONITORING
is not
TESTING

CANARIES

"Verification" in production



Verification in the wild Wild

Unit & Integration Tests

Property Based Testing

Fault Injection

Canaries

Research

Improving the Verification of Distributed Systems

Lineage Driven Fault Injection

'Cause I'm Strong Enough:
Reasoning about Consistency Choices in Distributed Systems

IronFleet:
Proving Practical Distributed Systems Correct

Towards Property Based Consistency Verification

Lineage-driven Fault Injection

Lineage-driven Fault Injection

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joshua Rosen
UC Berkeley
rosenville@gmail.com

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Failure is always an option; in large-scale data management systems, it is practically a certainty. Fault-tolerant protocols and components are notoriously difficult to implement and debug. Worse still, choosing existing fault-tolerance mechanisms and integrating them correctly into complex systems remains an art form, and programmers have few tools to assist them.

We propose a novel approach for discovering bugs in fault-tolerant data management systems: *lineage-driven fault injection*. A lineage-driven fault injector reasons *backwards* from correct system outcomes to determine whether failures in the execution could have prevented the outcome. We present MOLLY, a prototype of lineage-driven fault injection that exploits a novel combination of data lineage techniques from the database literature and state-of-the-art satisfiability testing. If fault-tolerance bugs exist for a particular configuration, MOLLY finds them rapidly, in many cases using an order of magnitude fewer executions than random fault injection. Otherwise, MOLLY certifies that the code is bug-free for that configuration.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—Distributed Databases

Keywords

fault-tolerance; verification; provenance

enriching new system architectures with well-understood fault tolerance mechanisms and henceforth assuming that failures will not affect system outcomes. Unfortunately, fault-tolerance is a *global* property of entire systems, and guarantees about the behavior of individual components do not necessarily hold under composition. It is difficult to design and reason about the fault-tolerance of individual components, and often equally difficult to assemble a fault-tolerant system even when given fault-tolerant components, as witnessed by recent data management system failures [16, 57] and bugs [36, 49].

Top-down testing approaches—which perturb and observe the behavior of complex systems—are an attractive alternative to verification of individual components. Fault injection [1, 26, 36, 44, 59] is the dominant top-down approach in the software engineering and dependability communities. With minimal programmer investment, fault injection can quickly identify shallow bugs caused by a small number of independent faults. Unfortunately, fault injection is poorly suited to discovering rare counterexamples involving complex combinations of multiple instances and types of faults (e.g., a network partition followed by a crash failure). Approaches such as Chaos Monkey [1] explore faults randomly, and hence are unlikely to find rare error conditions caused by co[illegible]plex combinations of failures. Worse still, fault injection techn[illegible]s—regardless of their search strategy—cannot effectiv[illegible] *coverage* of the space of possible failure scenarios. Fr[illegible]s such as FATE [36] use a combination of brute-force [illegible]d heuristics to guide the enumeration of faults; such heur[illegible]tic searc[illegible] strategies can be effective at uncovering rare failure scenarios, but

Molly In Action

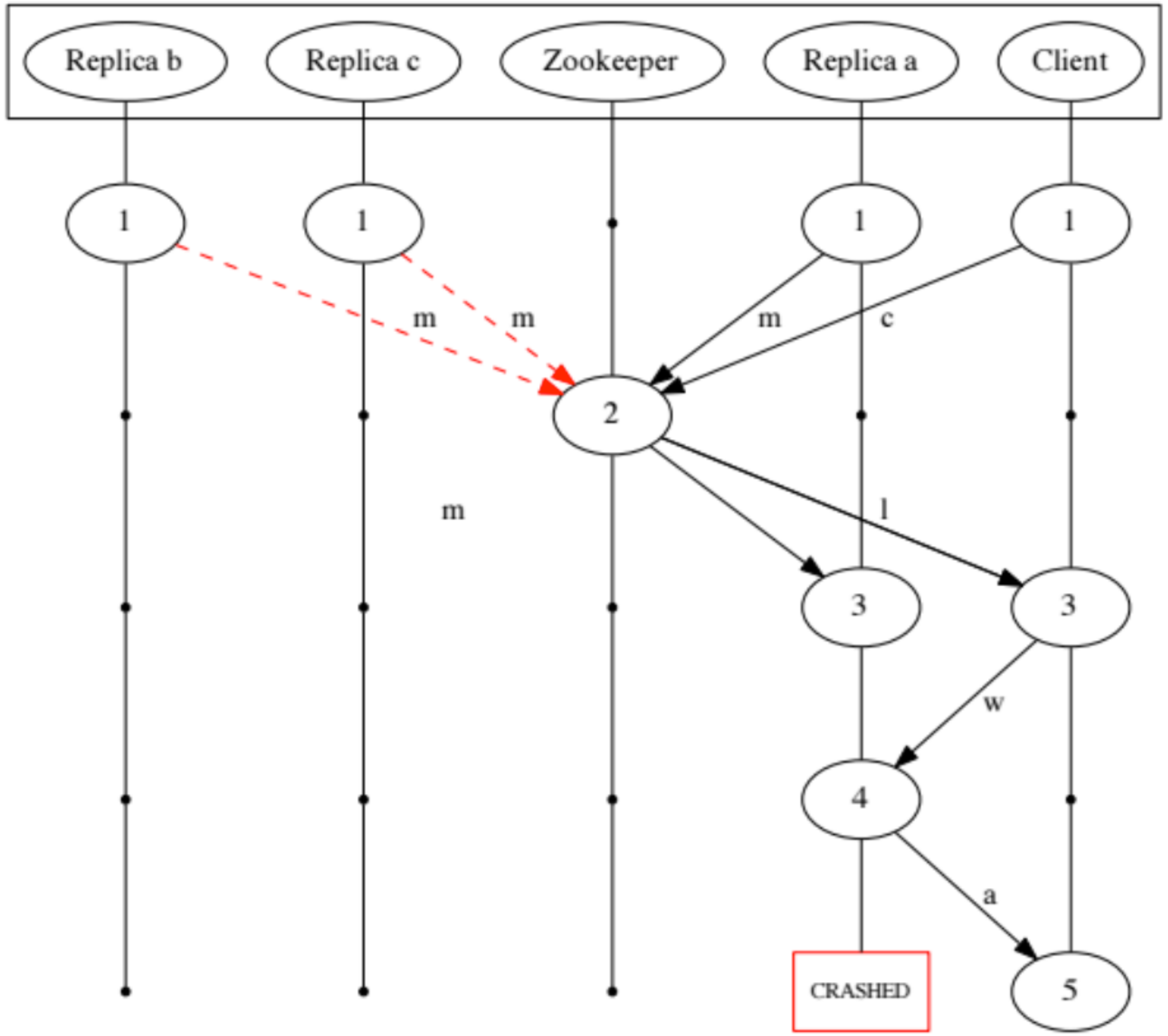
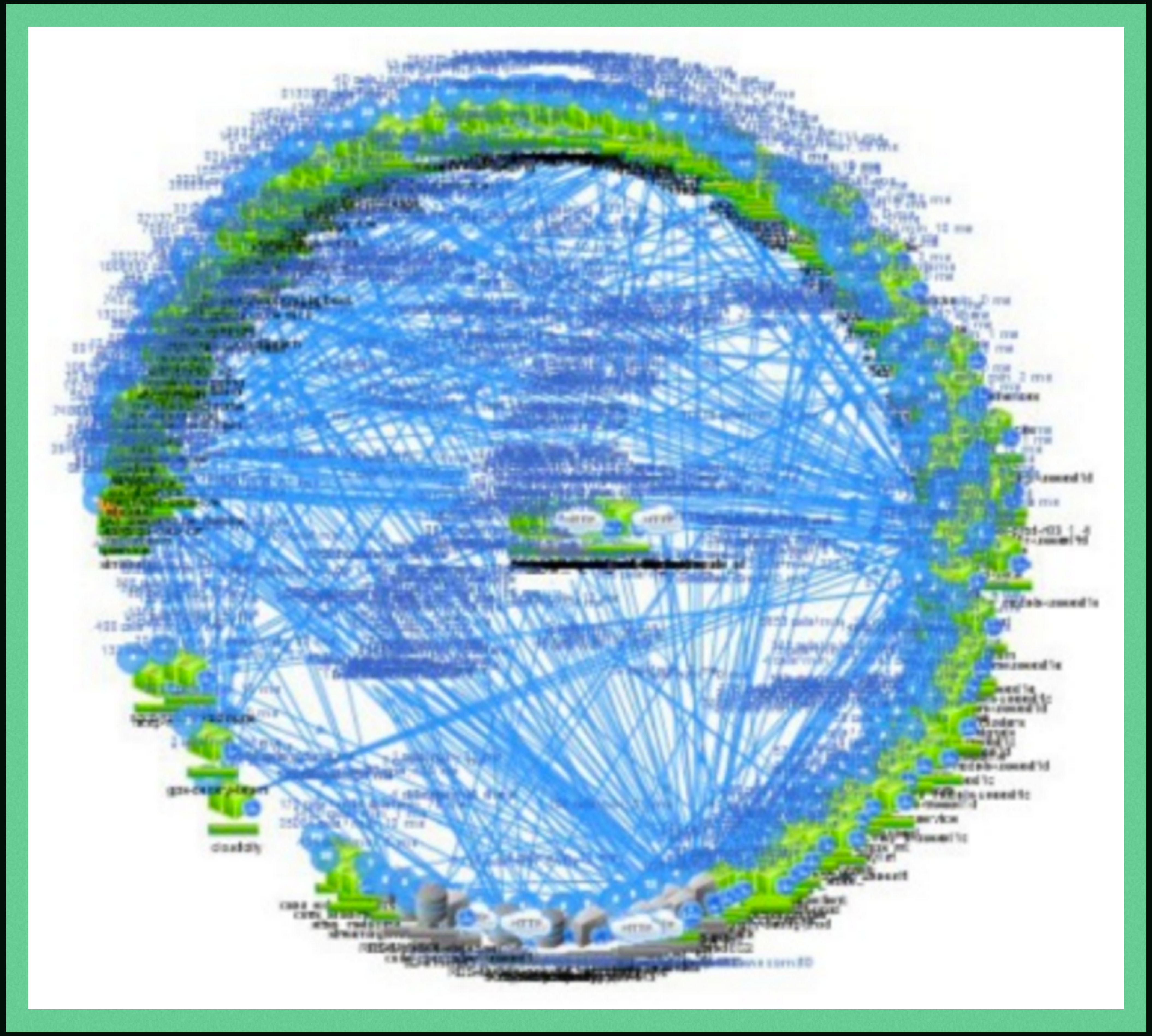


Figure 10: The replication bug in Kafka. A network partition causes b and c to be excluded from the ISR (the *membership* messages (**m**) fail to reach the Zookeeper service). When the client writes (**w**) to the leader a, it is immediately acknowledged (**a**). Then a fails and the write is lost—a violation of **durability**.

Lineage-driven Fault Injection

Netflix & Molly



Distributed Tracing + FIT To construct call graphs

Metric Systems to Determine if Call was a Success

Used FIT to Inject Failures determined by Molly

"Monkeys in Lab Coats": Applied Failure Testing Research at Netflix

Conclusion

Use Formal Verification on Critical Components

Unit Tests & Integration Tests find a multitude of Errors

Increase Confidence via Property Testing & Fault Injection


"Enjoy the ride, have fun, and test your freaking code"
Camille Fournier

Thank You

Peter Alvaro

Kyle Kingsbury

Christopher Meiklejohn

Alex Rasmussen

Ines Sombra

Nathan Taylor

Alvaro Videla

Questions

Resources:

http://github.com/CaitieM20/TheVerificationOfDistributedSystem

---

@caitie