The Verification of a Distributed System

A Practitioner's Guide to Increasing Confidence in System Correctness

Caitie McCaffrey

Distributed Systems Engineer

@caitie

caitiem.com

The Verification of a Distributed System

A PRACTITIONER'S GUIDE TO INCREASING CONFIDENCE IN SYSTEM CORRECTNESS

CAITIE MCCAFFREY

acm queue

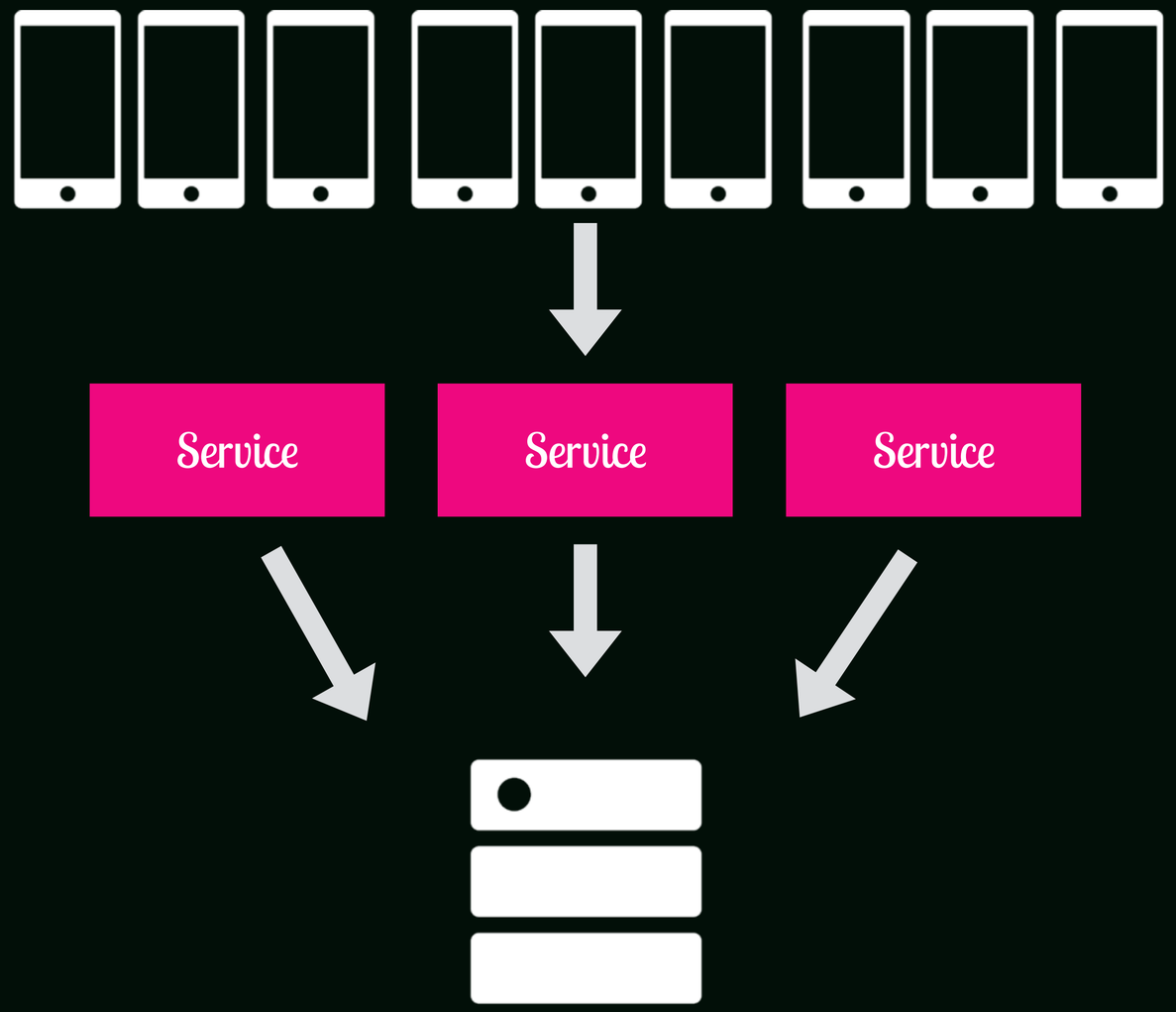
Leslie Lamport, known for his seminal work in distributed systems, famously said, "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." Given this bleak outlook and the large set of possible failures, how do you even begin to verify and validate that the distributed systems you build are doing the right thing?

Distributed systems are difficult to build and test for two main reasons: partial failure and asynchrony. Asynchrony is the nondeterminism of ordering and timing within a system; essentially, there is no now.[10] Partial failure is the idea that components can fail along the way, resulting in incomplete results or data.

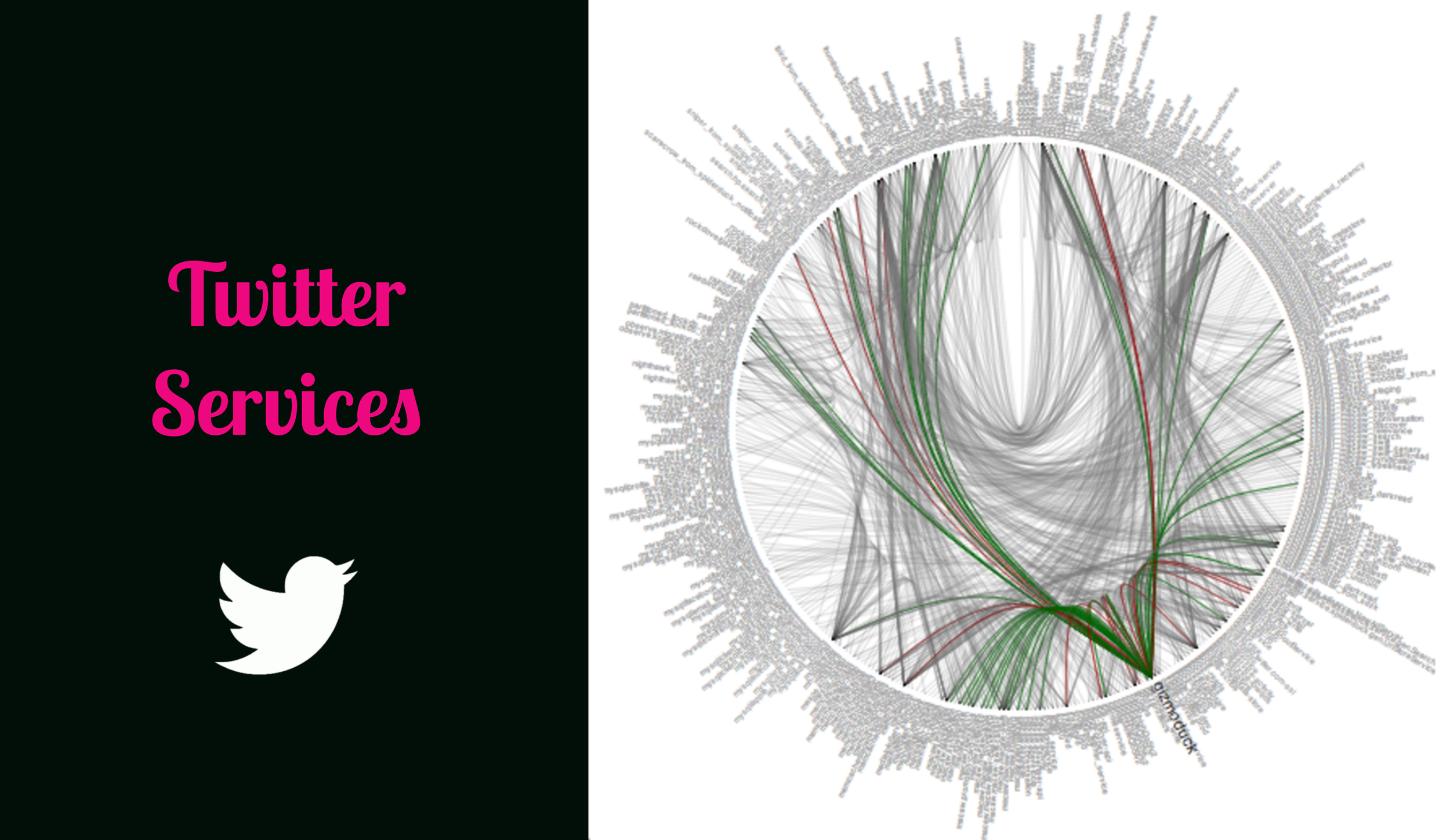
“A Distributed System is one in which the failure of a computer you didn’t even know existed can render your own computer unusable”

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/TheVerificationC

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

Formal Verification

Formal Specifications

Written description of what a system is supposed to do

TLA+  Coq

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 to avoid unnecessary complexity, but the essential complexity of the task remains high.

High complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts 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

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

Program Extraction

Planning for Change in a Formal Verification of the Raft Consensus Protocol

Doug Woos James R. Wilcox Steve Anton
Zachary Tatlock Michael D. Ernst Thomas Anderson
University of Washington, USA
{dwoos, jrw12, santon, ztatlock, mernst, tom}@cs.washington.edu

Abstract

We present the first formal verification of state machine safety for the Raft consensus protocol, a critical component of many distributed systems. We connected our proof to previous work to establish an end-to-end guarantee that our implementation provides linearizable state machine replication. This proof required iteratively discovering and proving 90 system invariants. Our verified implementation is extracted to OCaml and runs on real networks.

The primary challenge we faced during the verification process was proof maintenance, since proving one invariant often required strengthening and updating other parts of our proof. To address this challenge, we propose a methodology of planning for change during verification. Our methodology adapts classical information hiding techniques to the context of proof assistants, factors out common invariant-strengthening patterns into custom induction principles, proves higher-order lemmas that show any property proved about a particular component implies analogous properties about related components, and makes proofs robust to change using structural tactics. We also discuss how our methodology may be applied to systems verification more broadly.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Formal verification, distributed systems, proof assis-

of experience. Without the necessary tools to ensure the correctness of their systems, there is little hope of eliminating errors.

In previous work, we began to address this challenge by building Verdi [39], a framework for implementing and formally verifying distributed systems in the Coq proof assistant [9]. In this paper we describe our primary result to date using Verdi: the first formally verified implementation of the Raft [32] distributed consensus protocol.[1] The original Verdi paper discusses an implementation of Raft as a *verified system transformer*; Raft's correctness, the classic linearizability property, is expressed as correctness of a transformation from an arbitrary state machine to a fault tolerant system [39]. However, our previous proofs were focused only on phrasing linearizability as a VST correctness property; the proofs consisted of about 5000 lines and assumed several nontrivial invariants of the Raft protocol. This paper discusses the verification of Raft as a whole, including all the invariants from the original Raft paper [32]. These new proofs consist of about 45000 additional lines. Combining this with our previous proofs yields a complete proof that our Raft implementation is linearizable. Our effort yielded a verified implementation as well as insights into managing the verification process.

Raft ensures that a cluster of machines presents a consistent view of a state machine to the outside world, even in the presence of machine failures and unreliable message delivery. Broadly speaking, Raft provides similar functionality to the Paxos and Viewstamped Replication protocols [21, 29]. In practice, clusters using such al-

Chapar: Certified Causally Consistent Distributed Key-Value Stores

Mohsen Lesani Christian J. Bell Adam Chlipala
Massachusetts Institute of Technology, USA
{lesani, cjbell, adamc}@mit.edu

Abstract

Today's Internet services are often expected to stay available and render high responsiveness even in the face of site crashes and network partitions. Theoretical results state that causal consistency is one of the strongest consistency guarantees that is possible under these requirements, and many practical systems provide causally consistent key-value stores. In this paper, we present a framework called Chapar for modular verification of causal consistency for replicated key-value store implementations and their client programs. Specifically, we formulate separate correctness conditions for key-value store implementations and for their clients. The interface between the two is a novel operational semantics for causal consistency. We have verified the causal consistency of two key-value store implementations from the literature using a novel proof technique. We have also implemented a simple automatic model checker for the correctness of client programs. The two independently verified results for the implementations and clients can be composed to conclude the correctness of any of the programs when executed with any of the implementations. We have developed and checked our framework in Coq, extracted it to OCaml, and built executable stores.

Categories and Subject Descriptors C.2.2 [*Computer Communication Networks*]: Network Protocols—Verification; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness Proofs

Program 1 (p_1): Uploading a photo and posting a status

$0 \rightarrow$	**Alice**
$put(Pic, ☺)$;	▷ uploads a new photo
$put(Post, ☁)$	▷ announces it to her friends
$1 \rightarrow$	**Bob**
$post \leftarrow get(Post)$;	▷ checks Alice's post
$photo \leftarrow get(Pic)$;	▷ then loads her photo
$assert(post = ☁ \Rightarrow photo \neq \bot)$	

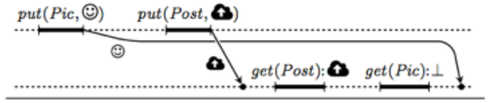


Figure 1. Inconsistent trace of Photo-Upload example

the downtime of a replica, other replicas can keep the service available, and the locality of replicas enhances responsiveness.

On the flip side, maintaining strong consistency across replicas [30] can limit parallelism [35] and availability. When availability is a must, the CAP theorem [19] formulates a fundamental trade-off between strong consistency and partition tolerance, and PACELC [3] formulates a trade-off between strong consistency

Program Extraction

Planning for Change in a Formal Verification of the Raft Consensus Protocol

Doug Woos James R. Wilcox Steve Anton
Zachary Tatlock Michael D. Ernst Thomas Anderson
University of Washington, USA
{dwoos, jrw12, santon, ztatlock, mernst, tom}@cs.washington.edu

Abstract

We present the first formal verification of state machine safety for the Raft consensus protocol, a critical component of many distributed systems. We connected our proof to previous work to establish an end-to-end guarantee that our implementation provides linearizable state machine replication. This proof required iteratively discovering and proving 90 system invariants. Our verified implementation is extr networks.
Th during the verification process was p ng one invariant often required strengthening a s of our proof. To address this challenge, we p y of planning for change during verification. M 's classical information hiding techniq ssistants, factors out common invariant-s o custom induction principles, proves hig w any property proved about a particul logous properties about related components, and makes proofs ro st to change using structural tactics. thodology may be applied to systems

Categories and Subject Descripto s F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Formal verification, distributed systems, proof assis-

of experience. Without the necessary tools to ensure the correctness of their systems, there is little hope of eliminating errors.

In previous work, we began to address this challenge by building Verdi [39], a framework for implementing and formally verifying distributed systems in the Coq proof assistant [9]. In this paper we describe our primary result to date using Verdi: the first formally verified implementation of the Raft [32] distributed consensus protocol.[1] The original Verdi paper discusses an implementation of Raft as a *verified system transformer*; Raft's correctness, the classic linearizability property, is expressed as correctness of a transformation from an arbitrary state machine to a fault tolerant system [39]. However, our previous proofs were focused only on phrasing linearizability as a VST correctness property; the proofs consisted of about 5000 lines and assumed several nontrivial invariants of the Raft protocol. This paper discusses the verification of Raft as a whole, including all the invariants from the original Raft paper [32]. These new proofs consist of about 45000 additional lines. Combining this with our previous proofs yields a complete proof that our Raft implementation is linearizable. Our effort yielded a verified implementation as well as insights into managing the verification process.

Raft ensures that a cluster of machines presents a consistent view of a state machine to the outside world, even in the presence of machine failures and unreliable message delivery. Broadly speaking, Raft provides similar functionality to the Paxos and Viewstamped Replication protocols [21, 29]. In practice, clusters using such al-

POPL 2016

"Our Verified Implementation is extracted to OCaml & runs on real networks"

★ COQ

Program Extraction

"We have developed & checked our framework in Coq, extracted it to OCaml, and built executable stores"

★ COQ

Chapar: Certified Causally Consistent Distributed Key-Value Stores

Mohsen Lesani Christian J. Bell Adam Chlipala
Massachusetts Institute of Technology, USA
{lesani, cjbell, adamc}@mit.edu

Abstract

Today's Internet services are often expected to stay available and render high responsiveness even in the face of site crashes and network partitions. Theoretical results state that causal consistency is one of the strongest consistency guarantees that is possible under these requirements, and many practical systems provide causally consistent key-value stores. In this paper, we present a framework called Chapar for modular verification of causal consistency for replicated key-value store implementations and their client programs. Specifically, we formulate separate correctness conditions for key-value store implementations and for their clients. The interface between the two is a novel operational semantics for causal consistency. We have verified the causal consistency of two key-value store implementations from the literature using a novel proof technique. We have also implemented a simple automatic model checker for the correctness of client programs. The two independently verified results for the implementations and clients can be composed to conclude the correctness of any of the programs when executed with any of the implementations. We have developed and checked our framework in Coq, extracted it to OCaml, and built executable stores.

Categories and Subject Descriptors C.2.2 [*Computer Communication Networks*]: Network Protocols—Verification; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness Proofs

Program 1 (p_1): Uploading a photo and posting a status

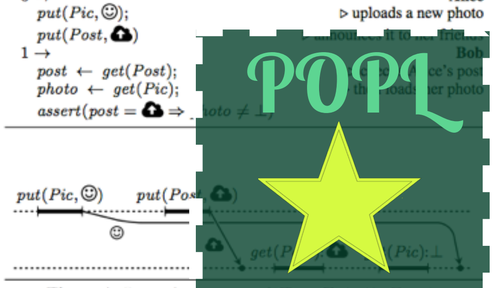


Figure 1. Inconsistent trace of

the downtime of a replica other replicas can keep the service available, and the locality of replicas enhances responsiveness.

On the flip side, maintaining strong consistency across replicas [30] can limit parallelism [35] and availability. When availability is a must, the CAP theorem [19] formulates a fundamental trade-off between strong consistency and partition tolerance, and PACELC [3] formulates a trade-off between strong consistency

POPL ★ 2016

Distributed Systems Testing in the Wild

"Seems Pretty Legit"

Unit Tests

Testing of Individual Software Components or Modules

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

77% of Production failures can be reproduced by a Unit Test

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

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

Three nodes or less can reproduce 98% of failures

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

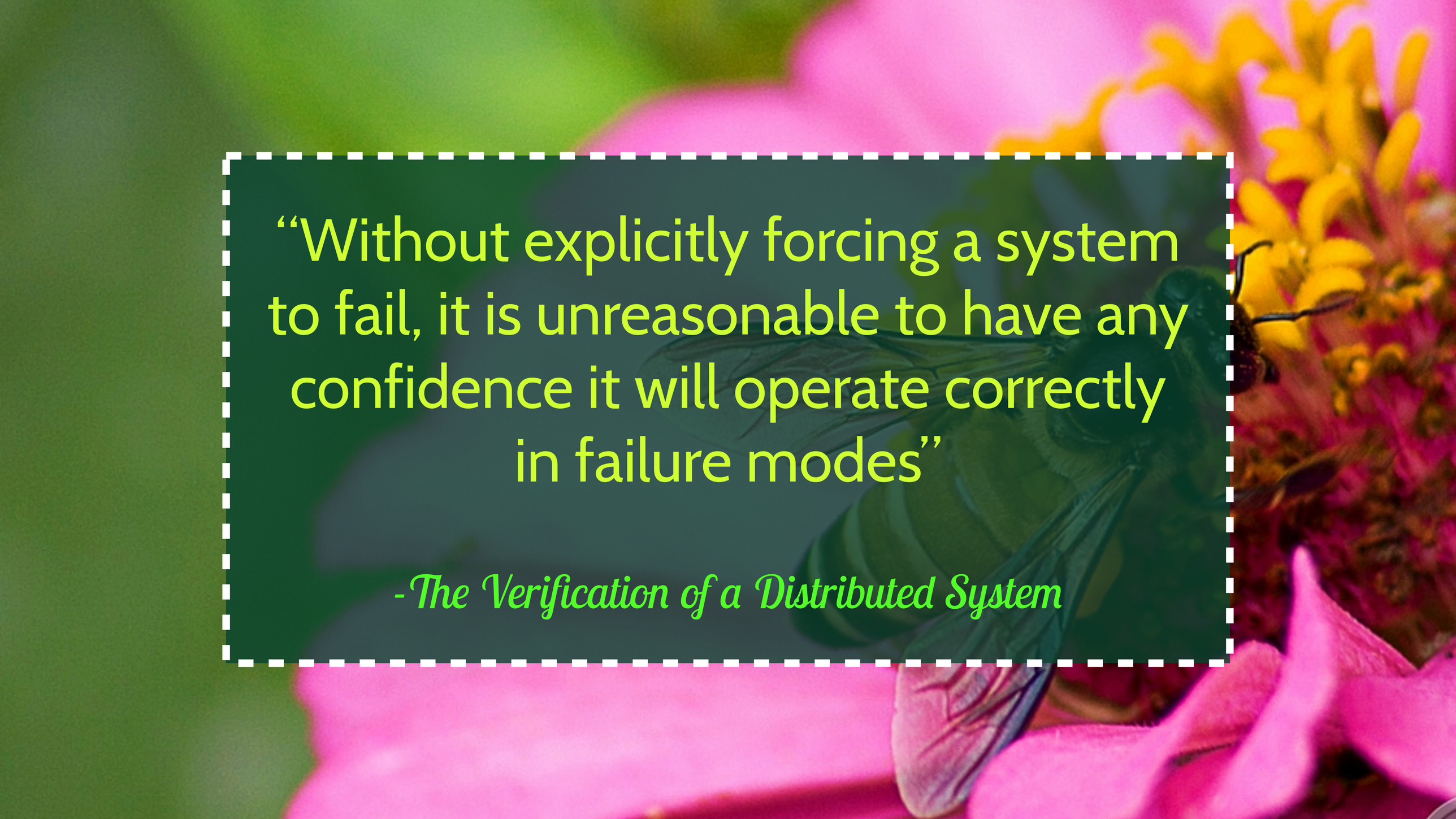
ScalaCheck Examples

```
import org.scalacheck._

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

```
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.

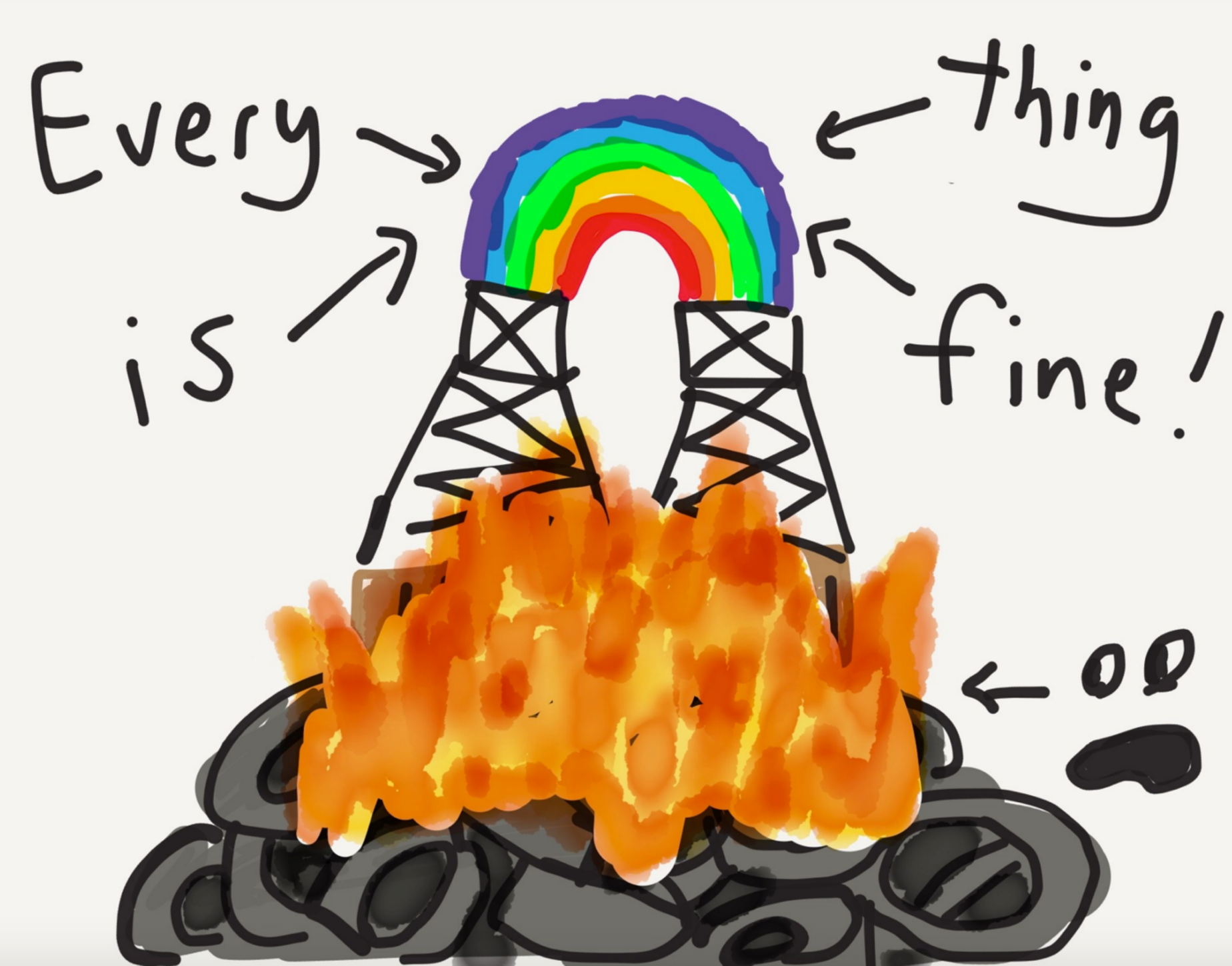
JEPSEN

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



credit: @aphyr

CAUTION: Passing Tests Does Not Ensure Correctness

GAME DAYS

Breaking your services on purpose

Resilience Engineering: Learning to Embrace Failure

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`

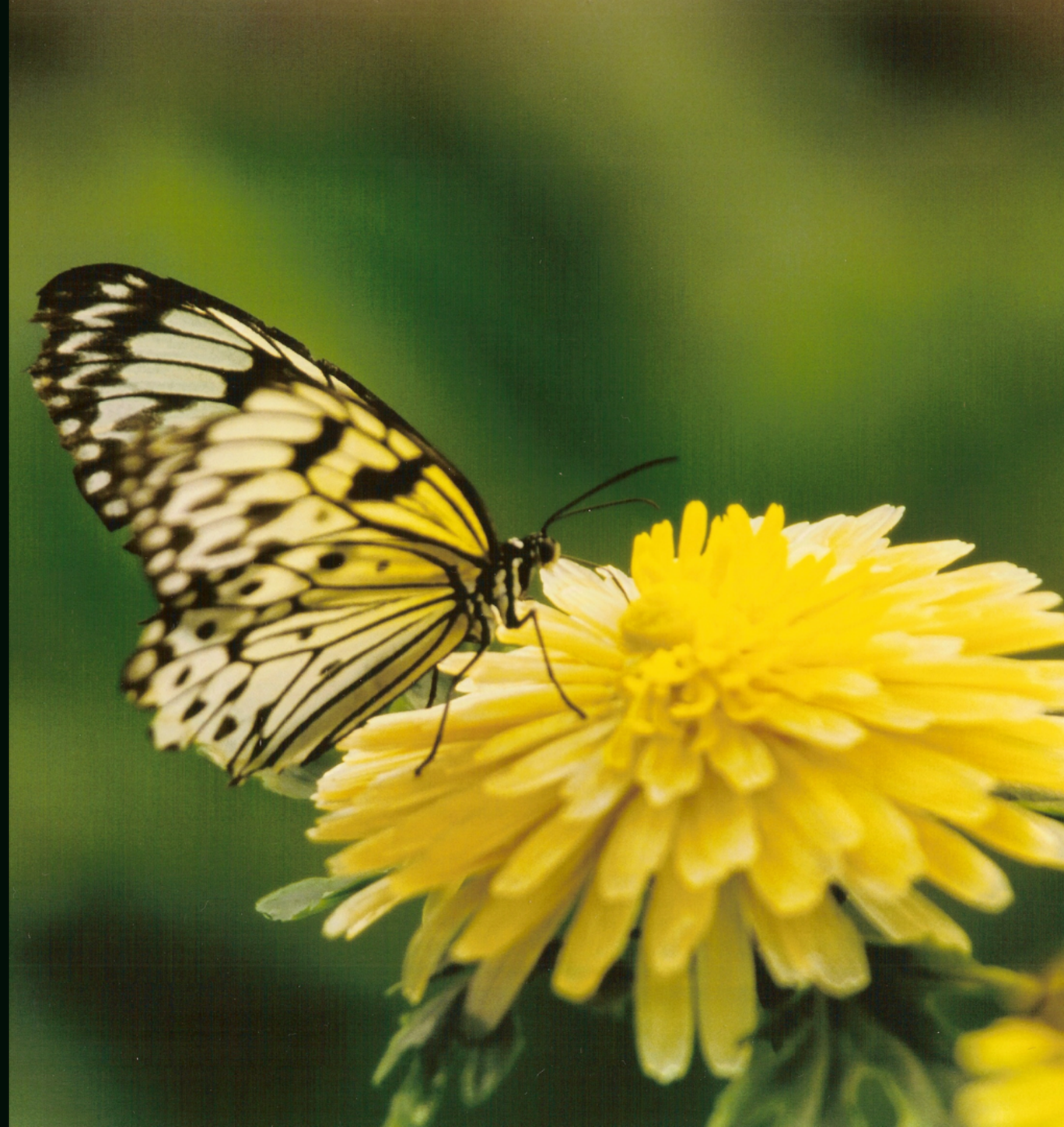
Some thoughts on

TESTING IN PRODUCTION

MONITORING
is not
TESTING

CANARIES

"Verification" in production



Verification in the Wild

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

POPL 2016

'Cause I'm Strong Enough

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

Alexey Gotsman
IMDEA Software Institute, Spain

Hongseok Yang
University of Oxford, UK

Carla Ferreira
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa, Portugal

Mahsa Najafzadeh
Sorbonne Universités, Inria,
UPMC Univ Paris 06, France

Marc Shapiro
Sorbonne Universités, Inria,
UPMC Univ Paris 06, France

Abstract

Large-scale distributed systems often rely on replicated databases that allow a programmer to request different data consistency guarantees for different operations, and thereby control their performance. Using such databases is far from trivial: requesting stronger consistency in too many places may hurt performance, and requesting it in too few places may violate correctness. To help programmers in this task, we propose the first proof rule for establishing that a particular choice of consistency guarantees for various operations on a replicated database is enough to ensure the preservation of a given data integrity invariant. Our rule is modular: it allows reasoning about the behaviour of every operation separately under some assumption on the behaviour of other operations. This leads to simple reasoning, which we have automated in an SMT-based tool. We present a nontrivial proof of soundness of our rule and illustrate its use on several examples.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Replication; causal consistency; integrity invariants

1. Introduction

To achieve availability and scalability, many modern distributed systems rely on *replicated databases*, which maintain multiple

use. Ideally, we would like replicated databases to provide *strong consistency*, i.e., to behave as if a single centralised node handles all operations. However, achieving this ideal usually requires synchronisation among replicas, which slows down the database and even makes it unavailable if network connections between replicas fail [2, 24].

For this reason, modern replicated databases often eschew synchronisation completely; such databases are commonly dubbed *eventually consistent* [47]. In these databases, a replica performs an operation requested by a client locally without any synchronisation with other replicas and immediately returns to the client; the *effect* of the operation is propagated to the other replicas only *eventually*. This may lead to *anomalies*—behaviours deviating from strong consistency. One of them is illustrated in Figure 1(a). Here Alice makes a post while connected to a replica r_1, and Bob, also connected to r_1, sees the post and comments on it. After each of the two operations, r_1 sends a message to the other replicas in the system with the update performed by the user. If the messages with the updates by Alice and Bob arrive to a replica r_2 out of order, then Carol, connected to r_2, may end up seeing Bob's comment, but not Alice's post it pertains to. The *consistency model* of a repli cated database restricts the anomalies that it exhibits. For the model of *causal consistency* [33], which we consider in th per, disallows the anomaly in Figure 1(a), yet can be implemer without any synchronisation. The model ensures that all replic the system see *causally dependent* events, such as the posts b Al- ice and Bob, in the order in which they happened. However, causal

```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Debit.class)
public class Account extends AnnotatedSchema {

    @XPR(value = {"Int amount", "Int balance" }, type = XPR.Type.ARGUME
    @XPR(value = "amount >= 0", type = XPR.Type.PRECONDITION)
    @XPR(value = "balance := balance + amount", type = XPR.Type.EFFECT)
    public static class Deposit extends AnnotatedOperation { }

    @XPR(value = {"Int amount", "Int balance" }, type = XPR.Type.ARGUME
    @XPR(value = {" amount >= 0 "}, type = XPR.Type.PRECONDITION)
    @XPR(value = "balance := balance - amount", type = XPR.Type.EFFECT)
    public static class Debit extends AnnotatedOperation { }
}
```

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

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