The Verification of a Distributed System

A Practitioner’s Guide to Increasing Confidence in System Correctness
Caitie McCaffrey
Distributed Systems Engineer

@caitie
caitiem.com
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. 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
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
Overview

Formal Verification
Provably Correct Systems

Testing in the Wild
Increase Confidence in System Correctness

Research
A New Hope
The Verification of a Distributed System

Abstract

Distributed Systems are difficult to build and test for two main reasons: partial failure & asynchrony. Testing and verifying these distributed systems must be addressed to create a correct system, and often times the resulting system has a high degree of complexity. Because of this complexity, testing and verifying these systems is critically important. This talk will discuss strategies for proving a system is correct, like formal methods, and less strenuous methods that can 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
MODULE HourClock

EXTENDS Naturals

VARIABLE hr

HCini == hr \in (1 .. 12)
HCnxt == hr' = IF hr \neq 12 THEN hr + 1 ELSE 1
HC == HCini \\ [][HCnxt] _hr

THEOREM HC => []HCini

Leslie Lamport, Specifying Systems
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 in the correctness of the system.
“Formal Methods Have Been a Big Success”

S3 & 10+ Core Pieces of Infrastructure Verified

2 Serious Bugs Found

Increased Confidence to make Optimizations

<table>
<thead>
<tr>
<th>System</th>
<th>Components</th>
<th>Line count (excl. comments)</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Background redistribution of data</td>
<td>645 PlusCal</td>
<td>Found 1 bug, and found a bug in the first proposed fix.</td>
</tr>
<tr>
<td>DynamoDB</td>
<td>Replication &amp; group-membership system</td>
<td>939 TLA+</td>
<td>Found 3 bugs, some requiring traces of 35 steps</td>
</tr>
<tr>
<td>EBS</td>
<td>Volume management</td>
<td>102 PlusCal</td>
<td>Found 3 bugs.</td>
</tr>
<tr>
<td>Internal</td>
<td>Lock-free data structure</td>
<td>223 PlusCal</td>
<td>Improved confidence. Failed to find a liveness bug as we did not check liveness.</td>
</tr>
<tr>
<td>distributed</td>
<td>Fault tolerant replication and reconfiguration algorithm</td>
<td>318 TLA+</td>
<td>Found 1 bug. Verified an aggressive optimization.</td>
</tr>
<tr>
<td>lock manager</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
“It’s a good idea to understand a system before building it, so it’s a good idea to write a specification of a system before implementing it”
“Formal methods deal with models of systems, not the systems themselves”

*Use of Formal Methods at Amazon Web Services*
Planning for Change in a Formal Verification of the Raft Consensus Protocol

Doug Woos, James R. Wilcox, Michael D. Ernst, and Steve Anton
University of Washington, USA
{dwoos, jrw12, santx, ztllock, merst, 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 linearizability on 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, factoring 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 [Formal verification and verification and Reasoning about Programs]: Mechanical verification

Keywords: Formal verification, distributed systems, proof assist-

Chaparral: Certified Causally Consistent Distributed Key-Value Stores

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

Abstract
Today’s Internet services are often expected to stay available and render high response times 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 Chaparral 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. This 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 (p1): Uploading a photo and posting a status

Alice
Bob

p1
post, photo

post

0

1

post ← get(Post)

get(phot0) ← photo

post ← get(Post)

sent(p1 to Bob) + photo (→)

get(Post) ← p1, photo

Program 2 (p2): Uploading a new photo

Alice
Bob

p2
newPhoto, photo

post

0

1

post ← get(Post)

photo ← get(phot0)

sent(p2 to Bob) + photo (→)

Figure 1. Inconsistencies trace of Photo-Upload Example
Planning for Change in a Formal Verification of the Raft Consensus Protocol

Doug Woos, James R. Wilcox, Zachary Tatlock, Steve Anton, Michael D. Ernst, Thomas Anderson
University of Washington, USA
{dwoos, jrw12, santon, ztt05, mene1, tomm}@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 connect our proof to previous work to establish an end-to-end guarantee that our implementation provides linearizability in state machine replication. This required iteratively discovering our verified implementation is networks. During the verification process we used one invariant often required form of our proof. To address this issue of planning for change during a classical information hiding assistants, factors out common and custom induction principles, which is any property proved about formalization that may be applied to systems.

Keywords: Formal verification, distributed systems, proof assistants, Coq, Cog.
“We have developed & checked our framework in Coq, extracted it to OCaml, and built executable stores.”
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

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 prevented if the bugs had been caught by our simple checking rules.

This 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 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 cause categorizations [33, 52, 50, 56], different types of causes including misconfigurations [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 the failures in a controlled environment.

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 prevented if the bugs had been caught by our simple checking rules.
77% of Production failures can be reproduced by a Unit Test

Simple Testing can Prevent Most Critical 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
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.
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
}

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.
Kyle has used this tool to show us that many of the Distributed Systems we know seem stable but are really just this. (cut to tire fire photo)

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

Every thing is fine

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 Game Day

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

Resilience Engineering: Learning to Embrace Failure
“During a recent game day, we tested failing over a Redis cluster by running \texttt{kill -9} on its primary node, and ended up losing all data in the cluster.”
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
'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, DL, 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 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 support multiple users and applications. Databases are a critical component of many modern systems, including the internet and enterprise applications. However, providing strong consistency guarantees for all operations in a system can be difficult and expensive.

In this paper, we propose a new approach to reasoning about consistency choices in replicated databases. We present a 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, allowing for 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.
Bank Application

Bank Account must be $> 0$

- Deposit Money
- Withdrawal Money
@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.ARGUMENT)
    @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.ARGUMENT)
    @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:

https://github.com/CaitieM20/Talks/tree/master/TheVerificationOfADistributedSystem

@caitie