UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title
Seeing the forest and the trees: Tackling Distributed Systems Problems by Querying Observations of Executions

Permalink
https://escholarship.org/uc/item/5cc3d139

Author
Ramasubramanian, Kamala

Publication Date
2022

Copyright Information
This work is made available under the terms of a Creative Commons Attribution-ShareAlike License, availalbe at https://creativecommons.org/licenses/by-sa/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

Seeing the forest and the trees: Tackling Distributed Systems Problems by Querying Observations of Executions

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Kamala Ramasubramanian

September 2022

The Dissertation of Kamala Ramasubramanian
is approved:

Associate Professor Peter Alvaro, Chair

Professor Ethan Miller

Assistant Professor Lindsey Kuper

Peter Biehl
Vice Provost and Dean of Graduate Studies

Copyright © by

Kamala Ramasubramanian

2022

Table of Contents

List of Figures v

List of Tables x

Abstract xi

Dedication xiii

Acknowledgments xiv

1 Introduction 1
- 1.1 Outline 4

2 Background 7
- 2.1 Assumptions 7
- 2.2 Requirements 8
  - 2.2.1 Understanding Fault tolerance behavior 9
  - 2.2.2 Troubleshooting distributed systems 9
  - 2.2.3 Identifying distributed systems behaviors 9

3 Understanding Fault Tolerance Under Protocol Evolution 11
- 3.1 Background and Motivation 13
- 3.2 Methodology and Results 15
  - 3.2.1 Catching Bugs Early 16
  - 3.2.2 Dormant bugs 17
  - 3.2.3 Optimizations 18

4 Understanding Fault Tolerance in Production Systems 21
- 4.1 Methodology 22
  - 4.1.1 Proof of equivalence 24
- 4.2 Evaluation 25

5 Troubleshooting: Debugging **29**
5.1 Background and Motivation 31
5.2 Methodology 34
5.2.1 Assumptions and Terminology 34
5.2.2 Correctness Specifications 36
5.2.3 Provenance Debugging Framework 37
5.2.4 Principal Strategies 38
5.3 Evaluation 43
5.3.1 Bug Taxonomy 46
5.3.2 Case Studies 47

6 Troubleshooting: Incident Localization **51**
6.1 Background and Motivation 54
6.1.1 Incident Study 54
6.1.2 Motivating Example 56
6.1.3 Limitations of existing approaches 59
6.2 Design & Methodology 64
6.2.1 Inputs and Outputs 65
6.2.2 System Overview 66
6.2.3 Application of ACT: An example 70
6.3 Evaluation 71
6.3.1 Determining the initial sample size 72
6.3.2 Experimental Methodology 74
6.3.3 Baseline techniques 77
6.3.4 Results 78
6.3.5 Integrating with Jaeger: Implementation Details 79
6.4 Iterative Localization 80

7 Identifying Distributed Systems Behaviors **85**
7.1 Methodology 87
7.2 Evaluation 90
7.3 Discussion 92

8 Conclusion **94**

Bibliography **98**

List of Figures

3.1 Concurrent-writes bug. Process M represents the master oracle; Process C represents the client; and all Process nN processes represent active replicas. 16
3.2 Concurrent-writes bug occurs in the single-write scenario as well . 17
3.3 Optimizing for sequence numbers 18

4.1 If the goal is that the data from at least one of the broadcasts is stored on either ReplicaA or ReplicaB, there are six ways this goal can be achieved. 22
4.2 Representation of the different ways a broadcast can fail to occur 23
4.3 Result from running fault tolerance experiments when the user is attempting to check out an item from an e-commerce site 26

5.1 Asynchronous primary/backup (“Async P/B”) replication protocol in Dedalus. Persistent relations in bold. 32

5.2 Simplified representation of the consequent provenance graph for a successful run of the Async P/B protocol from Figure 5.1 in reverse chronological order top to bottom. The message-passing events (postfixed `@async` in Figure 5.1) are colored turquoise. Consequent predicate `post` (lines 35–39 in Figure 5.1) is colored blue. Red-dashed vertices capture network connectivity to the respective other node. The two red gears hint at the computations that might not have taken place in a failed run, thus preventing the protocol from establishing the `post` predicate. 35
5.3 We assume our lineage-driven distributed debugger to be tightly integrated with an experiment selector providing the provenance graphs that form the basis of our analyses. A human operator applies the compiled suggestions. 36
5.4 Exemplary visualization of our three principal strategies for provenance-based debugging. Per strategy, equal vertex numbers identify the same logical event. Red boxes denote the result of operation `leaves` on a subgraph, blue boxes show the outcome of operation `reachable`. Orange-colored indices mark that the respective property evaluates to true for that vertex. 39

6.1 Percentage of impact by category - we have represented the top five of over a dozen different categories that emerged based on available data. Incidents arising due to breakdown in communication between components at the application level have the highest impact. 54
6.2 Typical incident timeline 56
6.3 This figure represents how SREs responded to an incident and the data sources they used (logs and metrics). The mitigation steps took SREs close to three hours, two and half of which was arriving at the correct mitigating action. 57

6.4 This is an idealized picture of graph differencing and contains only the relevant services. On the left is a partial view of a successful request where the token service was working as expected. On the right, we have the trace, after the restart of the payments service which continued to see errors due to incorrect firewall rules. . . . 57
6.5 Limitations of pairwise comparison - the two examples demonstrate the circumstances when pairwise comparison produces false alarms and can occur either separately or in combination. 61
6.6 Simple trace and an example of a view 65
6.7 ACT consists of applying three techniques: Symmetric difference, thresholding and reachability - in that order. 65
6.8 CDF of the number of traces to be sampled to identify any possible missing edge. The inlaid snippet of the CDF shows that a majority of calls can be identified with a sample that is orders of magnitude smaller. 73
6.9 For the cases when NodeCount and EdgeCount produce results, we plotted a CDF of number of results. ACT, meanwhile, produces exactly the expected answer for all of these cases. 73
6.10 CDFs of the number of results returned when we apply one or two techniques. Since the eBay dataset is noisy, symmetric difference and thresholding performs best, while symmetric difference and reachability generate the best results for DSB and HDFS. When all three techniques of ACT are applied, the result obtained is exactly the mitigation site. 73
6.11 CDFs of time taken to obtain a result. Reachability accounts for most of the time taken by ACT. Nodecount and Edgecount have highly variable time to result since trace of every unsuccessful execution needs to be compared with that of every successful execution and the number of unsuccessful executions can vary widely. 76

6.12 Iterative localization cycles through projection, filtering, and localization. We use ACT for localization, but projection and filtering can produce different results depending on the choice of fields to project down to and the results of prior localization. Two such examples are shown here. 81
6.13 Shows the sequence of results produced by iterative localization for an example bug in the fallback path - when db-primary fails, db-secondary is invoked, but the call fails because of the lack of write permissions. The result of subsequent localizations are informed by and improve upon results of prior localizations. Legend: Dashed lines represent calls. Gray nodes represent a service or service:operationname that was in a successful execution but not in an unsuccessful execution; blue nodes represent the reverse. Finally, a green dashed line indicates a successful call while red indicates an unsuccessful call. 82
6.14 Shows that even when iterative localization does not streamline results, it can add specifics that help engineers take action. In this case, the bug is that the call to requestmapper was critical but not recognized as such and an RPC failure from app-server to requestmapper caused failure of the request as a whole. Legend: Dashed lines represent calls. Gray nodes represent a service or service:operationname that was in a successful execution but not in an unsuccessful execution 83

7.1 (a) represents common design patterns such as fallbacks and caching effects, where the red and green arrows represent failed and successful calls respectively. The dotted lines represent a service to which a call was attempted, but the message was dropped or lost in transit. (b) is an example trace taken from a real production system . . . 87

7.2 System workflow showing the steps in our methodology with a running example. The id in the mappings corresponds to identity, which means that the fields are retained as-is. index_of indicates that the start time is converted into a logical time and we have also shown how status code is mapped to one of three strings. 89

List of Tables

5.1 Taxonomy of 52 distributed concurrency bugs from the TaxDC collection and the asynchronous primary/backup protocol from Figure 5.1. Legend: ✔ = yes, ✘ = no, ❍ = it depends. 44

6.1 This table explains how we simulate the three failure modes we consider. For each, we describe the input, how traces are mutated and the expected output. We also specify the conditions that need to be satisfied in each case for a trace to be mutated. All mutated traces represent unsuccessful executions. 74

6.2 ACT computes exactly the expected result for all but a few cases. In contrast, NodeCount and EdgeCount produce wrong answers for 30-50% of simulations. Answer = Set of localizations returned, Exact Answer = Answer is minimal, Superfluous Answer = Answer subsumes expected result, Wrong Answer = Answer does not contain expected result, No Answer = Answer is the null set. . . . 75

7.1 Instances of patterns in different applications 91

Abstract

Seeing the forest and the trees: Tackling Distributed Systems Problems by Querying Observations of Executions

by

Kamala Ramasubramanian

Distributed systems are ubiquitous but continue to be challenging to understand, build, and troubleshoot. Fundamentally, reasoning about distributed system behaviors is hard due to the effects of partial failures and nondeterminism in system executions. For example, we expect systems to remain available even if some number of replicas fail. These problems are exacerbated by the dynamic nature and scale of production systems today. Tooling support has lagged behind the pace at which systems are being deployed, urgently requiring more research in this space.

Our overarching claim is that many common distributed systems problems such as improving fault tolerance or debugging failures can be addressed by querying observations of executions. Since our system view consists of observations of system executions, rather than the system itself, we require that executions must exercise varied paths to enable us to derive useful insights about the system. A second requirement is that since events in distributed executions may be separated by space and time, observations must capture both events and how they relate to each other within individual executions.

Our key insight is that we need to aggregate information from many executions while preserving the causal relationships within individual executions to answer the posed questions. We use provenance graphs (a growing area of research) and distributed traces, which have seen increased adoption in industry, as observations of system executions since they capture the causality of event interactions within executions and normalize them to aggregate information across many executions.

Prior work uses observability infrastructure to aggregate information from many executions or compares pairs of executions while preserving causal relationships within executions, but not both. Methodologies to address problems such as fault detection, localization and anomaly detection [1–5] based on aggregating logs and metrics have been explored. Other work compares pairs of executions [6–8] for interactive debugging, performance diagnostics, workload and capacity modeling. The former approach either disregards the causality of event interactions within executions or attempts to infer them [9–12], producing sub-par results, while the latter is lacking since it only considers a single pair of executions but many varied execution paths are exercised.

In our work, we have developed and evaluated techniques for understanding and improving fault tolerance behavior, troubleshooting systems, and identifying instances of common design patterns that have applications in building domain knowledge, feature development and debugging performance issues. We explore how the problems that can be solved are constrained differently or change entirely depending on factors such as the granularity and format of system observations, timeline of expected response, how interactive (or not) techniques are expected to be, and the level of detail in the result produced.

To my family

Acknowledgments

My work was possible in large part due to the support of my family and guidance of my advisors. First and foremost, thanks to mom and dad, Bharathi and Ram, for always believing in me and supporting me throughout this journey. My sister, Kanchana, supported me when even I doubted myself. My husband, Erik, had my back at all times and baked me cookies to keep me going! My parents-in-law, Bill and Ruth Cornelius, have been nothing but supportive form the get-go.

Next, I'd like to thank my advisor, Peter Alvaro, for being my mentor in this journey. Our early work together really set the tone for my PhD and greatly influenced my dissertation research. I'd also like to thank Lindsey Kuper and Ethan Miller for being on my advancement as well as dissertation reading committees. Their feedback and inputs changed the light in which I considered the problems I addressed and significantly improved the quality of my writing. Many thanks to Jonathan Mace, who invited me to intern at MPI-SWS and also served on my advancement committee. Jonathan helped me define my problem succinctly and in doing so, enabled me to take the first steps towards devising ACT to effectively localize incidents.

Thanks to Ashutosh Raina for the many brainstorming sessions and discussions, especially when working on fault tolerance testing and ACT in collaboration with eBay. Thanks to Eliana Philips for being willing to discuss and try out some of the more far-fetched ideas when we were working together on mining microservice design patterns - I really appreciate your enthusiasm and attention to detail. Boaz Leskes from Elasticsearch facilitated my research in understanding fault tolerance as protocols evolve and contributed to our paper while Kathryn Dahlgren, Asha Karim, Sanjana Maiya and Sarah Borland co-wrote the “Growing a Protocol” paper with me. Thank you to everyone in Disorderly Labs as well as in the

broader LSD lab for their generous and valuable feedback.

My research was supported by grants from the National Science Foundation and by eBay. Special thanks to Ravi Punati and the entire SRE team at eBay who took time to discuss incident management in production systems and made themselves available to answer questions. Last, but not the least, I want to thank Tracie Tucker, Alicia Haley, and the entire Graduate Advising team, who have helped me navigate the various stages of the PhD program.

Thank you - I could not have done this without you all!

Chapter 1

Introduction

Happy families are all alike; every unhappy family is unhappy in its own way

Anna Karenina by Leo Tolstoy

Querying observations of system executions is an effective way to tackle distributed systems problems. In this work, we develop techniques that ask questions of observations of system executions - captured as-is without modifying the system in any way - and use their answers to reason about the underlying system. We then use insights from such reasoning to understand, improve, and troubleshoot distributed systems.

Asking and answering questions is a time-honoured way of gathering information. Today, different categories of end users pose questions and use the answers to make decisions about implementing, operating or debugging the system in question. We present a few example questions posed in various contexts. Designers may ask verification related questions such as *"What do all successful executions have in common?"* to characterize the behavior of successful executions. Knowledge of system behavior can be used to inform good test design as well as to test that optimizations to the system uphold desired correctness properties [13].

"What faults do I have no evidence my system will tolerate?" is a question verification experts may ask when trying to understand the fault tolerance properties of the system. For a predefined class of faults, one way to understand fault tolerance properties of a system is via fault-injection testing - checking that the system behaves correctly while systematically injecting faults. Some examples of fault injection frameworks include Chaos Monkey [14], for randomized fault injection, and Lineage Driven Fault Injection [15], which uses observability of system executions to drive experiment selection. Ordering fault scenarios would reduce the number of experiments to be run, making fault tolerance testing more efficient.

A developer debugging an outage might want to know - *"How do successful and unsuccessful executions differ?"* to identify events and causal relationships that provide actionable insights. Many engineers use incomplete and outdated mental models and tools based on observations that provide a siloed view of the system to answer the above question. Answering the above question automatically requires effective data representation and appropriate comparison operators. Some previous approaches [7, 16] that compare observations of executions to answer this question have had limited success since their data representation lost essential information with respect to interactions amongst system events.

To identify instances of design patterns in a running system, a developer may ask: *"What can successful executions teach us about how they succeed?"* While design templates for common behaviors such as fallbacks and caching are industry standard and common knowledge, asking the above question allows us to discover instantiations of these templates. Discovering possible instantiations has a wide range of applications in building domain knowledge, feature development, and debugging both behavioral and performance issues. Other examples of questions that may be posed are *"Is this execution anomalous?"* for anomaly detection

or *"Where are slow executions spending a disproportionate amount of time?"* to address performance regressions.

Our key insight is that, to answer the posed questions *automatically*, we need to develop techniques that reason about events and how they relate to each other within individual executions as well as in aggregate across executions. Since events in distributed system executions can be separated in space and time, causal relationships between events in individual executions can be used to deduce system behavior. However, an observation from a single execution represents only one of many possible executions. To incorporate information about different execution paths as well as to paper over incomplete data, techniques need to reason in aggregate across executions. Aggregation requires that we normalize system observations i.e. that we only consider fields that are consistent across executions. We typically project down to consider only service names or file:line numbers.

Since traces and data provenance capture events within individual executions and how they relate to each other, we choose to capture observations of system executions as traces or provenance graphs. Traces provide a request-level view of the system. For a given user request, a trace captures its events and how they relate to each other. The most common representation for a trace is as a directed acyclic graph (DAG), whose nodes represent events and edges represent the interactions between different events. Provenance [37–42] is well established in database and systems literature for providing explanations of outcomes and can be used to obtain causality at the granularity of system records. Other observability signals include metrics, logs, and events.

Prior work that aggregates information uses logs or metrics to determine trends in system behavior or uses machine learning for fault detection, localization, and debugging [1, 2, 4, 5]. Such techniques either attempt to *infer* causality of in-

teractions or disregard them completely. Work that takes into account causal interactions within executions typically only compares pairs of executions [6–8], which may produce irrelevant or incorrect results.

Putting the details together, we automate solving distributed systems problems by developing techniques that reason both within individual executions and in aggregate across traces or provenance graphs obtained by witnessing many executions. In this thesis, we focus on understanding fault tolerance behavior, troubleshooting, and identifying behaviors of distributed systems. Next, we outline the chapters and for each, we discuss the high-level problem and our contributions.

1.1 Outline

Chapter 2 discusses the assumptions we make about the systems under study as well as the class of bugs we focus on finding and fixing. We also briefly discuss the implications of our assumptions on understanding fault tolerance and troubleshooting. In the next four chapters, we present work on understanding fault tolerance and troubleshooting systems. For each of these problems, we present two systems. One system produces record level data provenance while the other is integrated with distributed traces. We contrast the high-level problems that can be solved and the techniques uniquely suited to each case.

In Chapters 3 and 4, we focus on understanding and improving the fault tolerance of systems via fault injection testing. Our approach attempts to intelligently choose fault scenarios that could drive the system into an undesirable state by reasoning about observed executions.

- (Chapter 3) Modeling the data replication protocol at Elasticsearch [13]: We modeled checkpointed versions of the data replication protocol in a specification language (Dedalus) that produces record-level data provenance.

Contributions: Via this experience, we demonstrate the necessity of checking the fault tolerance space of a system for each change and how we can efficiently do so with our approach.

- (Chapter 4) Fault tolerance testing at eBay: For two payment workflows at eBay, we explored the fault tolerance behavior of the system by employing end to end tests that produce distributed traces for injected fault scenarios.

  Contributions: To explore fault scenarios optimally when the number of possible scenarios is large, we developed a formulation that returns the most likely fault scenario given domain specific probabilities.

In Chapters 5 and 6, we focus on fixing bugs that arise from machine crashes and message drops.

- (Chapter 5) Nemo [47], our prototype debugger: Nemo uses record level data provenance to provide assistance by pointing to a line or region of code or generates repairs to achieve correct behavior.

  Contributions: To generate repairs in addition to providing assistance with debugging, we extend differential reasoning using provenance graphs of executions to co-analyze the provenance of correctness specifications. We also provide a new taxonomy of real-world distributed systems bugs that our techniques apply to.

- (Chapter 6) ACT now: Aggregate Comparison of Traces for Incident Localization [48]: Our incident localization framework takes as input distributed traces and produces actionable insights for engineers to take action.

  Contributions: To overcome the challenges of using traces from production systems, we have developed techniques that compare sets of traces rather than pairs of traces which enables automated and effective incident local-

ization. We have also integrated ACT with Jaeger, an open source tracer, to enable online comparison of sets of traces.

In Chapter 7, we discuss nascent work identifying instances of common design patterns by querying observations of system executions. This work has a wide range of applications and has produced promising preliminary results. Chapter 8 concludes and outlines future directions for work.

Chapter 2

Background

To answer the how and why questions of distributed systems executions, we exploit observations of system executions. Specifically, we use observations in the form of data provenance and distributed traces. In this chapter, we first discuss assumptions we make about the systems under study. We also discuss the class of bugs we focus on finding and fixing and the additional requirements systems need to meet for our techniques to be applicable to address a given problem.

2.1 Assumptions

We predicate our work on three main assumptions. First, we assume processes communicate via message passing and therefore, system observations capture some, but not necessarily all, causal relationships for a given execution. Distributed traces and data provenance are represented as graphs where nodes correspond to events and edges between two nodes indicate a causal relationship between them. Causality captured in graphs represents happens-before relationships in system executions.

Our second assumption concerns correctness of system executions. Traces can-

not be used to establish correctness since they may not have enough information to do so. For example, correctness properties involving system state cannot be checked using traces since they do not capture distributed state. Rather, executions are tagged as successful or unsuccessful based on external success criteria that serve as a proxy for correct system behavior. We make stronger assumptions when using data provenance. In addition to being able to determine if a given execution is successful or not, we assume that the program and the correctness guarantees are written in the same specification language.

Finally, we assume that non-determinism in system executions arises from non-deterministic effects in the environment and not from programs. Some examples of such environmental effects include randomization of instance selection, partial failures and network delays. We use this assumption when we expect executions that exercise the same functionality, given the same inputs and schedule, to take similar paths through the system. As a result, our techniques are not suited to reason about systems that incorporate randomized algorithms.

2.2 Requirements

We focus on finding and fixing bugs characterized by omission faults. In the omission fault model, processes may crash and messages may be dropped. To be characterized by an omission fault implies that a particular fault combination, once found to trigger a bug, always triggers the bug and is not dependent on environmental factors such as timing or network delays. We do not reason about concurrency bugs that are triggered by non deterministic scheduling orders.

As discussed previously, there are many possible execution paths and system executions exercising the same functionality often vary due to non-determinism in the environment. Therefore, it is necessary to aggregate information by witnessing

observations from a large number of executions to reason about overall system behavior. Next, we outline requirements specific to each of the problems for which we have developed techniques to address.

2.2.1 Understanding Fault tolerance behavior

Since the space of inputs is large, to focus computational resources on fault selection, we require that inputs to programs being checked be pre-selected. Secondly, to explore faults that include process crashes and message drops, we require record-level provenance to be captured. Record-level data provenance captures the processes and messages in a system execution while distributed traces only capture the processes explicitly while the messages are implicitly captured based on the calls between processes.

2.2.2 Troubleshooting distributed systems

In our work, troubleshooting distributed systems takes two forms - debugging and incident localization. Our techniques for debugging use provenance graphs as inputs to provide assistance and in some cases generate repairs. To make this possible, we not only require correctness specifications to be written down in the same specification language as the program, but further require that they are written down as implications of the form "if <condition holds>, then <corresponding expected behavior>".

2.2.3 Identifying distributed systems behaviors

We analyze observations of successful executions to identify possible instantiations of well-understood design patterns. In some cases, we need to compare observations from two or more executions. We require that the analyzed observa-

tions reflect the effects of only one change to the system. In our setting, a single change translates to failure of an RPC call or service instance crash. This requirement is necessary since different changes can interact with each other leading to false positives that we cannot disambiguate. Although false positives are still possible, more deterministic executions with higher quality system observations will produce fewer false positives.

In this chapter, we have made explicit our assumptions when using data provenance and distributed traces as system observations. We have also outlined the class of bugs we solve for as well as additional requirements systems need to meet for each of the problems we consider. These constrain the problem space and we will refer back to them in future chapters.

Chapter 3

Understanding Fault Tolerance Under Protocol Evolution

For a system to be fault tolerant, it should behave correctly for every combination of some predefined class of faults. Correctness can be defined using criteria such as if the system upholds safety guarantees, is functional and available, produces anticipated results for different classes of inputs, or if system functionality degrades gracefully in the face of failures. We consider omission faults in our work - where processes may crash and messages may be dropped.

One approach to gaining confidence that a system works correctly under faults is via fault injection testing. Fault injection testing methodologies answer the how of fault injection. Approaches such as ChaosMonkey [49] and Gremlin [50] actuate random failures and check if the system behaves as expected. Other approaches [51,52] simulate fault scenarios in requests and avoid the cost of actuating failures in the system. The fault space of a system is exponential in the number of system components even if we restrict ourselves to crash faults only. Therefore, an exhaustive search of the fault space is intractable.

Lineage Driven Fault Injection [15], which reasons about the fault tolerance

behaviors of systems via fault injection testing by asking the question: "What do successful executions have in common?", represents the closest related work. We have built upon this work in two ways: a) We have demonstrated that the methodologies developed can be applied iteratively to different versions of the protocol as it evolves and by so doing, we have shown the necessity of fault tolerance testing after every change to the program. and b) We have developed a formulation for determining the most likely fault scenario that we should explore next based on probabilities of failure associated with components.

We evaluate the fault tolerance behavior of two systems - one that produces provenance for its runs while the other is integrated with distributed tracing:

- We have modeled Elasticsearch's data replication protocol (similar to primary-backup) in Dedalus (an extension of Datalog) that produces data provenance for its runs. We discuss how we evolved the protocol in detail in this chapter.

- In our work at eBay, we considered end to end tests corresponding to two payment workflows and attempted to drive the system into an undesirable state by intelligently choosing fault combinations. This is the focus of the next chapter.

We combine insights from observing system executions when injecting faults to reason about the fault tolerance behavior via two questions : 1.) What do observations of successful executions tell us about the fault tolerance of the system? and 2.) What fault scenario should we explore next? While we need to answer both questions when reasoning about observations and determining the next fault scenario to explore, we will focus on the first question in this chapter and on the latter in the next. Section 3.1 motivates the problem while Section 3.2 provides an overview of our methodology and describes the main results and takeaways.

3.1 Background and Motivation

Common distributed systems wisdom warns us *never to reinvent.* If we have a problem requiring consensus, we use Paxos [53] (or Raft [54]); if we need strong consistency data replication for availability, we use Primary/Backup [55] or Chain Replication [56]. To disseminate updates, we use reliable broadcast [57]. 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 implementers 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, both 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.

As a simple example, in a three node system with one primary and two followers, the system should behave identically when either of the followers crash. That is, we expect identical behavior for the two fault scenarios F1 crash and F2 crash, where F1 and F2 represent Follower1 and Follower2 respectively. Needing to consider a class of inputs means that we need to perform a principled search of the space of execution schedules on every commit, but an exhaustive search of the space of possible combinations of faults is intractable. To guide our exploration of

the fault space, we use LDFI [15] - an analysis and fault selection framework that builds a model of the system based on good system executions and only explores fault scenarios that can potentially force the system to a bad state.

3.2 Methodology and Results

Record-level data provenance may reveal multiple ways for the protocol to achieve correct behavior in a single run. Only a fault scenario that falsifies all of these will either reveal additional information - a new way for the protocol to succeed - or force the system to a bad state.

The core Elasticsearch data replication protocol is a variation of primary backup. All client requests are routed to the primary and a request is acknowledged only after the primary receives acknowledgements from 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 our 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 further 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.

Figure 3.1: Concurrent-writes bug. Process M represents the master oracle; Process C represents the client; and all Process nN processes represent active replicas.

3.2.1 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 3.1, 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.

Discovering the bug requires primary failure after launching successful writes to only a subset of backup replicas. Furthermore, a replica from the unlucky sub-

set 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.

3.2.2 Dormant bugs

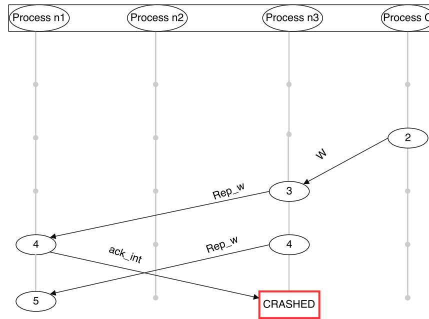


Figure 3.2: Concurrent-writes bug occurs in the single-write scenario as well

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 3.2 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

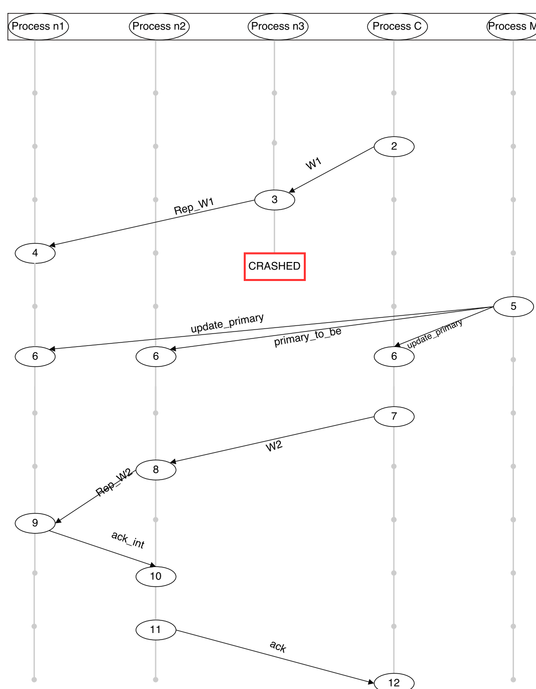


Figure 3.3: Optimizing for sequence numbers

input data. This brings into sharp focus the fact that the input data we start with matters in finding interesting bugs.

3.2.3 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.

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 3.3 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.

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.

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.

To summarize, 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.

In this chapter, we have described 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 tools like LDFI are 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. In the next chapter, we discuss our experiences exploring the fault tolerance behavior of production systems and how we navigate this space, which may be much larger, efficiently.

Chapter 4

Understanding Fault Tolerance in Production Systems

For many large systems, functional and end-end tests that validate system functionality are checks of system correctness. Distributed traces are the state of the art for production systems today rather than the more fine grained record level provenance. Given end-to-end tests associated with an external success measure, we want to ensure that for every combination of faults (in this case, crash faults), the behavior of the system is correct. Due to the scale of systems under consideration, the number of fault scenarios that can lead to a potentially bad state can be many. Therefore, we developed a formulation to order fault scenarios by their likelihood of occurrence based on domain specific probabilities as a way to navigate the (potentially) large space of fault scenarios.

In section 4.1, we describe the intuition of our approach, present the underlying mathematical formulations and prove the equivalence between different formulations to find the most likely fault scenario efficiently by taking advantage of advances in Integer Linear Programming (ILP) solvers. In Section 4.2, we discuss the results of using our methodology for two payment workflows at eBay

and compare them with that of a randomized fault injection framework, such as ChaosMonkey [49].

4.1 Methodology

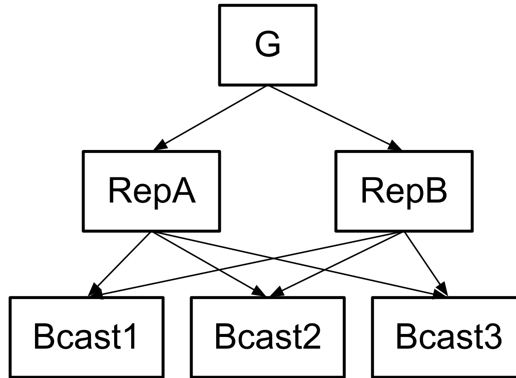


Figure 4.1: If the goal is that the data from at least one of the broadcasts is stored on either ReplicaA or ReplicaB, there are six ways this goal can be achieved.

Consider Figure 4.1 as an example. To falsify the goal that data from at least one of the broadcasts is stored on either ReplicaA or ReplicaB, one element in each set needs to have failed.

```
{RepA OR Bcast1} AND {RepB, Bcast1} AND
{RepA OR Bcast2} AND {RepB OR Bcast2} AND
{RepA OR Bcast3} AND {RepB OR Bcast3}
```

The highlighted solutions in Figure 4.2 represent fault scenarios, either of which could potentially force the system into a bad state. Our formulation answers the question of which fault scenario we should explore *next*.

By observing that every fault scenario has to contain an element from each set, we can formulate the problem as the minimal hitting set problem. To order fault

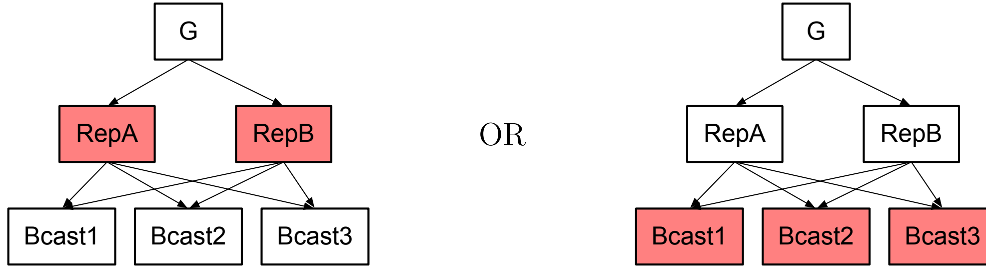


Figure 4.2: Representation of the different ways a broadcast can fail to occur

scenarios by increasing cardinality from the smallest to the largest, the minimal hitting set problem can be formulated as follows:
Given a finite set S and a collection $P \equiv \{P_1, \ldots P_n\}$ where each $P_i \subset S \ \forall i \in 1 \ldots n$, the minimum hitting set of P is a set $H \subseteq S$ such that for all $i \in 1, \ldots n$,

$$P_i \cap H \neq \emptyset \tag{4.1}$$

$$\nexists H', H' \subset H \wedge P_i \cap H' \neq \emptyset \tag{4.2}$$

That is, H is the smallest set of elements from S that "hits" every $P_i \in P$ by intersecting with it on at least one element.The minimal hitting set formulation produces exactly the same results as before but orders the solutions by increasing cardinality. It is equivalent to set cover and known to be NP-Hard [58].

To take advantage of state-of-the-art optimizations for integer linear programming solvers, we re-formulate the problem as an optimization problem. To do so, we define following additional notation. For all $j \in 1 \ldots |S|$, let X_j correspond to the jth element of S according to some arbitrary ordering. X_j is interpreted as an integer that is either 0 or 1 - intuitively, it is 1 if is is "selected" as being part of H. Let M_{ij} be 1 if the variable j appears in P_i, and 0 otherwise. The minimal hitting set problem is now written as the following ILP problem:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{j=1}^{|S|} X_j \\
\text{subject to} \quad & \sum_{i=1}^{n} M_{ij} X_j \geq 1
\end{aligned}
\tag{4.3}
$$

Solutions to our formulation return fault scenarios by increasing cardinality. The scenario with the *smallest number* of elements may not be the *most likely* to occur, however. We incorporate domain specific failure probabilities to find the *most likely* fault scenario. If the probability of failure is equal for all events, the solutions returned would be identical to those returned in the minimal hitting set formulation. To order the fault scenarios by likelihood of occurrence, we re-write our objective function as follows:

$$
\begin{aligned}
\text{maximize} \quad & \prod_{j=1}^{|S|} Pr(j)^{X_j} \\
\text{subject to} \quad & \sum_{i=1}^{n} M_{ij} X_j \geq 1
\end{aligned}
\tag{4.4}
$$

where $Pr(j)$ is the probability of failure of the event corresponding to the jth element of S. We rewrite it so the optimization is linear and prove that the two forms are equivalent.

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{|S|} x_j log(Pr(j)) \\
\text{subject to} \quad & \sum_{i=1}^{n} M_{ij} X_j \geq 1
\end{aligned}
\tag{4.5}
$$

4.1.1 Proof of equivalence

Both formulations (4.4 and 4.5) share the same constraints, only the objective functions are different. We denote the objective function of 4.4 as f_1 and that of 4.5 as f_2. Each X_j is either 0 or 1 and $Pr(j)$ denotes the probability of failure

(ranging between 0 and 1) of the event corresponding to the jth element. We show that the formulations in 4.4 and 4.5 return the same solutions in the same order.

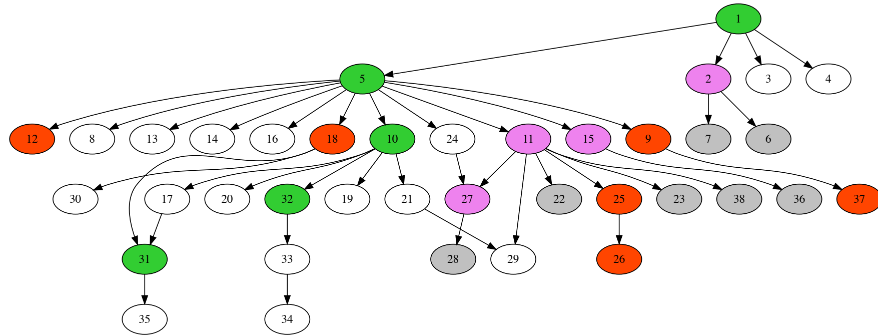
Let s_1 be a solution that maximizes f_1. We show that s_1 is the first solution for f_2 as well. Suppose not. Suppose there is another solution s_2 that maximizes f_2. Then, $2^{f_2(s_2)} > 2^{f_2(s_1)}$, which in turn implies that $f_1(s_2) > f_1(s_1)$. This is a contradiction since we know that s_1 is a solution that maximizes f_1.

The converse can be shown similarly. Let s_2 be a solution that maximizes f_2. We show that s_2 is also maximizes f_1. Suppose not. Suppose there is another solution s_1 such that $f_1(s_1) > f_1(s_2)$. Then $log(f_1(s_1)) > log(f_1(s_2))$ which implies that $f_2(s_1) > f_2(s_2)$. We have arrived at a contradiction since s_2 is a solution that maximizes f_2.

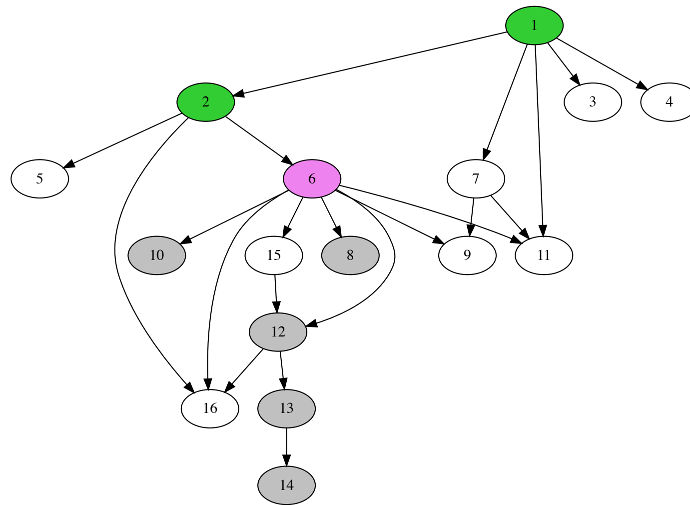
In our work, we set the probability of failure of a service to be proportional to the rank of the service in the trace and iteratively generated fault scenarios. Using distance of a service from the root of the DAG as a proxy for its impact mimics how engineers think, making it a realistic measure. That is to say, a fault scenario involving a service close to the root of the trace is more likely to drive the system into an unexpected state than one that is far away from the root. Secondly, exploring fault scenarios that involve services close to the root of traces first enables us to prune large sections of the fault space if they are tolerated, reducing the number of experiments that need to be run.

4.2 Evaluation

We carefully select tests to exercise end to end functionality when checking out items at eBay. We test two workflows - users adding items to a shopping cart(CreateCart) and completing the purchase(MakePurchase). Each of these represents a complete user action and produces a trace for every action. Traces



(a) Bugs found in the path corresponding to adding item(s) to a cart



(b) No bugs found in the path corresponding to making the purchase

Figure 4.3: Result from running fault tolerance experiments when the user is attempting to check out an item from an e-commerce site

touch many services and we employ our formulation with traces as input to determine likely fault scenarios to explore.

To find the next fault scenario to explore, we take as input traces from all previous successful runs of a given test. Using our formulation, we associate probabilities with different services and find the most likely fault scenario. We then inject the suggested fault and re-run the test. Either, the test run is successful

and we collect another trace from a successful test run or we have discovered a bug. We run tests repeatedly until we have exhausted fault scenarios and there are no bugs or we discover a bug.

To present our results succinctly, we unified the nodes and edges seen in individual traces into a single graph. These are represented in Figure 4.3. The green nodes represent services that are to be excluded from fault tolerance testing, the pink nodes are services in which a fault was injected and found to be tolerated, the gray nodes are services in which we do not need to inject faults based on the observed graphs and the red nodes are services in which we discovered bugs.

In the CreateCart workflow, we discovered 6 bugs (Figure 4.3a), all of which we found in a few hours. These range from a known product issue for which a bug had already been raised, unexpected behavior and incorrect status code reporting. In the MakePurchase workflow, we found no bugs (Figure 4.3b) and by ordering the fault scenarios, we performed only 7 experiments in total. Not finding bugs is an important result as it tells us when we can stop fault tolerance testing.

As a baseline, we wrote a fault injector that simulates injecting faults uniformly at random. The probability associated with every fault scenario is the same, unlike with our approach. As the number of services in the system increases, the space of possible scenarios expands exponentially, making it unlikely that we will hit upon a likely fault scenario by random exploration. To find the same bugs that we did, the random fault injector would take an order of magnitude more number of experiments. The confidence interval for the number of experiments needed is $[157, 264]$, with a confidence level of 0.95 - calculated via the t-test.

Our results demonstrate that a prioritized approach to fault tolerance testing that biases towards reducing the space of fault scenarios is useful when testing workflows using end-end tests. Ordering fault scenarios helps reduce the fault

space to be explored by explicitly using causality of interactions to determine the optimal fault scenario to be explored based on the witnessed traces. Ordering is achieved by re-posing the problem as an optimization problem and iteratively solving it. Our results from running experiments are promising since we were able to complete both the experiments within a few hours for both workflows.

We have described both the formulation we developed to determine the most likely fault scenario as well as how we use system observations to reason about the fault tolerance of systems. Our principled approach to exploring the fault space has found bugs in two different systems and more importantly, has concluded that there are no more experiments to try in one payment workflow. The latter is an important result which fosters confidence in systems since exhaustive testing is intractable. In the next two chapters, we discuss how we can effectively troubleshoot distributed systems.

Chapter 5

Troubleshooting: Debugging

In this chapter and the next, we focus on troubleshooting distributed systems for bugs that are triggered by faults such as crashes and message loss. Specifically, we consider debugging - the act of finding and fixing a bug - and incident localization - identifying the location, for eg. a component, where engineers can take action to restore the system. For both problems, the end goal is for the system to behave correctly.

Correct system behavior can be defined in a variety of ways. Examples include system upholding safety properties, being functional and available, producing anticipated results for different classes of inputs or the functionality degrading gracefully in the face of failures. Oftentimes, the success or failure of executions is determined by an external criteria such as a HTTP request returning 200, a credit card being successfully charged, page pieces being correctly loaded, etc. Engineers employ such an external success criterion to automatically mark executions as successful or unsuccessful.

To troubleshoot systems, asking how can lead us to answering the why. We take as inputs observations of successful and unsuccessful executions and reason about failures by asking: How do successful and unsuccessful executions differ?

By asking the above question, we developed two systems that aid troubleshooting:

- A prototype debugger, Nemo [59], that uses record level data provenance to provide assistance by pointing to a line or region of code or generate repairs to achieve correct behavior. We will discuss Nemo in depth in this chapter.

- ACT, our incident localization framework, that takes as input distributed traces and produces actionable insights for engineers to take action. ACT is the focus of Chapter 6.

Debugging involves finding and fixing a line or region of code as a result of which failures arise. For failures involving an incorrect transition, a faulty configuration line, or an off-by-one loop bound, identifying a line or region of code to modify helps fix it. Failures that are a result of insufficient synchronization or redundancy typically require the program to add rules to specify the protocol more completely. We refer to these as *commission* and *omission* failures respectively. Our taxonomy for 52 bugs from large-scale distributed systems [60] shows the prevalence of commission and omission failures.

Recent work has shown the use of data provenance in debugging distributed systems [44, 46, 61], fault localization [62] and network diagnostics [45, 61, 63, 64]. We build upon prior work and have developed techniques that provide assistance and can even generate repairs when the correctness specifications are written in the same provenance-enhanced language as the program. We generate repairs by by co-analyzing the provenance of the system state with the provenance of the specification predicates. Because the specification describes the non-distributed behavior of the program, it guides the generation of code changes that correct the distributed program towards compliance with its sequential specification.

The rest of the chapter is organized as follows: In Section 5.1, we present a motivating example that serves as a running example throughout the chapter.

In Section 5.2, we discuss our assumptions and principal strategies. In Section 5.3, we provide a new taxonomy for 52 real-world distributed bugs from large-scale distributed systems [60], determining for each whether our framework can suggest program corrections or provide debugging assistance. Then, we discuss our results using Nemo to repair errors of omission and identify root causes of errors of commission in six protocol implementations, of which we discuss four in case studies.

5.1 Background and Motivation

To motivate our approach, we start with a simple, "buggy" protocol implementation. Figure 5.1 shows a programmer's first attempt at implementing primary/backup replication [65] in the declarative programming language Dedalus [66]. Dedalus is a subset of Datalog [67] with negation that also allows for reasoning about programs' behavior over time. Starting with the inputs, data flows through defined relations in increasing logical time steps until no further updates are possible. @next and @async represent *when* data propagates between relations if the necessary conditions are met - @next implies propagation at the next time step, while @async implies propagation at some future time step. Once all updates are made, the correctness specifications are evaluated to determine if the system behaves as expected for the given inputs.

In our example, a single "primary" node accepts requests to write data items, disseminates them to passive "backup" nodes, and ultimately responds to clients. The correctness specification for primary/backup is shown in lines 33–39 of Figure 5.1. If a payload was marked as acknowledged in table `acked` at the client (*antecedent* predicate `pre`, lines 33–34), then it *must* appear in the `log` of all non-crashed nodes in the system (*consequent* predicate `post`, lines 35–39). In any

```
1  // Initially, client Cli sends request Pload to
2  // primary node Prim via the network (@async).
3  request(Prim, Pload, Cli)@async :-
4      begin(Cli, Pload),
5      conn_up(Cli, Prim);
6
7  // Asynchronous version of primary/backup:
8  // On receipt of a request, the primary immedi-
9  // ately sends an acknowledgment to the client.
10 // Clients persist acknowledgments.
11 ack(Cli, Prim, Pload)@async :-
12     request(Prim, Pload, Cli);
13 acked(Cli, Prim, Pload) :-
14     ack(Cli, Prim, Pload);
15
16 // The primary replicates received requests
17 // in background to all replicas Rep.
18 replicate(Rep, Pload, Prim, Cli)@async :-
19     request(Prim, Pload, Cli),
20     replica(Prim, Rep);
21 // Primary and all replicas write received
22 // requests durably to local storage.
23 log(Prim, Pload) :-
24     request(Prim, Pload, Cli);
25 log(Rep, Pload) :-
26     replicate(Rep, Pload, _, _);
27
28 // Correctness specification:
29 // As soon as a client received an acknowl-
30 // edgment for its request (pre),
31 // the request's payload is durably stored
32 // on all alive nodes (post).
33 pre(Pload) :-
34     acked(Cli, Prim, Pload);
35 post(Pload) :-
36     log(Node, Pload),
37     primary(Prim, Prim),
38     notin crash(Node, Node, _),
39     Node != Prim;
```

Figure 5.1: Asynchronous primary/backup ("Async P/B") replication protocol in Dedalus. Persistent relations in bold.

run where this is not the case the correctness expectation is violated (details in Section 5.2.2). The rest of the protocol works as: the primary accepts requests from clients (`request`, lines 3–5) and replicates them to all replicas (`replicate`, lines 18–20), which store them durably in their local state (`log`, lines 23–26).

Unfortunately, the programmer has tried to optimize this protocol for performance. Lines 11–14 show that an acknowledgment for a request is sent from primary to client immediately when it was received. Primary crash or loss of replication messages could prevent the request from becoming durable despite having been acknowledged at the client!

Suppose the programmer found the bug during a test and was able to reproduce it. The laborious process of finding its root or proximal causes has only just begun. Conventional debugging approaches like `grep`'ing through logs from all nodes or attaching legacy debuggers to each are no help at all, as this protocol-level bug arises not on individual nodes per se, but in their interactions across space and time. Distributed provenance [15, 46, 68] stitching together node-local views into explanations of how data transited a distributed system seems a more appropriate tool for this kind of debugging. Abstracting from details specific to the collection process, in Figure 5.2 we show a provenance graph explaining how a tuple marking establishment of predicate `post` was computed in a successful run of the protocol from Figure 5.1. Unfortunately, even the trivial motivating protocol presented here produces in total a set of provenance graphs with 280 vertices and 205 edges, making it impractical to debug by staring at them.

Differential provenance by Chen et al. [61] refines provenance to specifically aid in root cause analysis. By automatically visualizing the difference between a successful and a failed provenance graph, it allows users to quickly identify key events that differentiate between an observed failed and a known successful run. Unfor-

tunately, while differential provenance has been shown to help highlight critical errors in configuration, input data, and even program logic (i.e., the *presence* of a mistake), the bug in our replication protocol has no such smoking gun. Rather, it is the *absence* of necessary synchronization that makes the protocol fail to uphold its contract - there is no bad line or tainted data to point to.

Readers familiar with replication protocols know how to work around the problem: the primary has to postpone client acknowledgment until after confirmation from backups. Implementing this fix, however, requires more than finding and fixing an incorrect program statement - something is *missing* and needs to be added. We appear to be at an impasse. We cannot debug the program by comparing successful and failed runs, because the successful runs provide no hint about how to fix the fundamental problem. Instead, the programmers need to rethink the program's logic. Or do they? In this work, we provide evidence that we are able to *generate* corrections for these kind of problems in a great many cases.

5.2 Methodology

We begin this section by reviewing the assumptions we make for our strategies to be effective and introduce necessary terminology. We then describe the query language and capabilities of our provenance debugging framework.

5.2.1 Assumptions and Terminology

We expect the distributed system under inspection to operate in the *omission fault model*, in which messages may - independently - be delayed arbitrarily long, be lost, and processes may fail by crashing. We assume the system to consist of at least two processes that communicate via messages and have access to storage

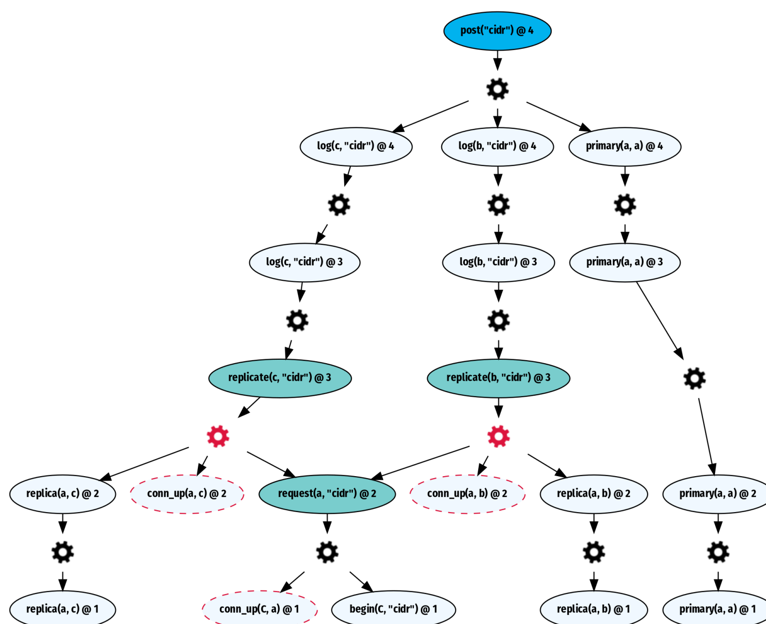


Figure 5.2: Simplified representation of the consequent provenance graph for a successful run of the Async P/B protocol from Figure 5.1 in reverse chronological order top to bottom. The message-passing events (postfixed `@async` in Figure 5.1) are colored turquoise. Consequent predicate `post` (lines 35–39 in Figure 5.1) is colored blue. Red-dashed vertices capture network connectivity to the respective other node. The two red gears hint at the computations that might not have taken place in a failed run, thus preventing the protocol from establishing the `post` predicate.

that is durable across restarts. As input to our strategies, we expect a collection of *provenance graphs* from a series of *runs* of the program. Figure 5.2 represents one such provenance graph for the consequent of a successful run of the protocol from Figure 5.1 reduced in detail to show the structure of expected graphs. In case we identify a reproducible violation of the correctness specification (a bug), it is going to be the last run which we thus call *failed*. All others are *successful* runs produced under different schedules, message orderings, or faults. A program with at least one failed run is *buggy*, otherwise it is *correct*. We assume to be operating in concert with an experiment selector that generates these graphs (e.g., integration

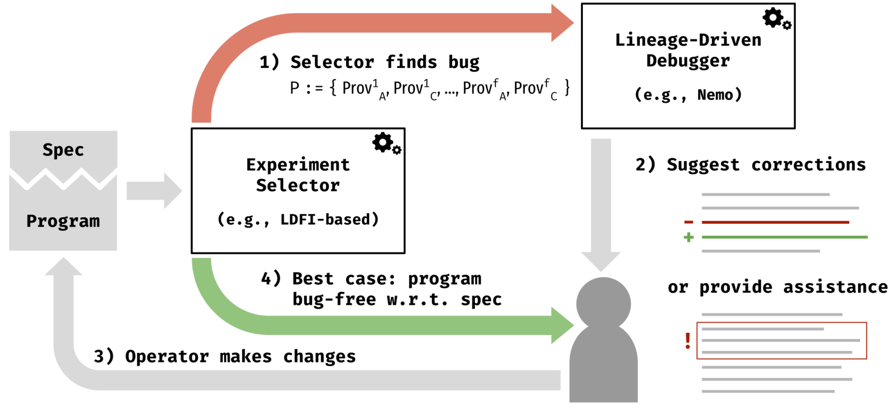


Figure 5.3: We assume our lineage-driven distributed debugger to be tightly integrated with an experiment selector providing the provenance graphs that form the basis of our analyses. A human operator applies the compiled suggestions.

tests). In practice, we imagine this to be a tight loop, such as the layout visualized in Figure 5.3: the selector identifies a bug, the bug is fed into our strategies where corrections are generated, and an operator attempts to apply the suggestions. Repaired programs are resubmitted to the selector until all bugs are resolved.

5.2.2 Correctness Specifications

Any verification solution expects that a system under test be accompanied by a description of what it means to be correct. We require correctness specifications in the form of *implications*, $\mathcal{A} \rightarrow \mathcal{C}$, where antecedent $\mathcal{A}$ and consequent $\mathcal{C}$ are first-order logic formulae over the set of relations comprising the system's distributed state. *Invariants* such as "account balance is positive" can be captured in $\mathcal{C}$ with $\mathcal{A}$ set to true. $\mathcal{C}$ must thus hold in all runs, as we would expect of an invariant. Many distributed correctness properties, however, are not bare invariants. Due to the possible faults in distributed systems, there exist runs in which properties that require communication are never achieved. A reliable broadcast protocol disseminating a message to a group of nodes will never succeed if all nodes or

the network stop functioning. Thus, distributed correctness properties are most commonly expressed as implications where $\mathcal{A}$ holds when the run is not vacuously correct and $\mathcal{C}$ then enforces expected distributed behavior.

Put differently, $\mathcal{A}$ is true when a possible good state is achieved and $\mathcal{C}$ describes the state that, given $\mathcal{A}$, must occur. For example, the specification for reliable broadcast reads: "If a correct process delivers a message ($\mathcal{A}$), then all correct processes deliver it ($\mathcal{C}$)". Agreement safety in commit protocols could say: "If a participant commits (aborts) a transaction ($\mathcal{A}$), then all participants commit (abort) ($\mathcal{C}$)". Durable replicated data stores require: "If a write is acknowledged at the client ($\mathcal{A}$), then it is durably stored on all alive replicas ($\mathcal{C}$)".

For our strategies, the program under test and its correctness specification are expressed in the same logic programming language. As part of program state, records of $\mathcal{A}$ (`pre` in Figure 5.1) and $\mathcal{C}$ (`post` in Figure 5.1) are enriched with provenance describing how they occurred. Every record in $\mathcal{A}$ comes with an explanation why the run that produced it was not vacuously correct, while every record in $\mathcal{C}$ provides an explanation why the run upheld the property of interest.

5.2.3 Provenance Debugging Framework

The debugging strategies presented here manipulate the set of provenance graphs $\mathcal{P}$ from the runs of the distributed program under inspection. Elements of $\mathcal{P}$ are directed acyclic graphs describing the provenance for $\mathcal{A}$ or $\mathcal{C}$ of run $i = 1, \ldots, n$. Members of $\mathcal{P}$ are called $\text{Prov}^i_{\mathcal{A}}$ or $\text{Prov}^i_{\mathcal{C}}$, depending on their role in the specification. For one successful and one failed run this amounts to $\mathcal{P} = \{\text{Prov}^1_{\mathcal{A}}, \text{Prov}^1_{\mathcal{C}}, \text{Prov}^2_{\mathcal{A}}, \text{Prov}^2_{\mathcal{C}}\}$. In short, $\mathcal{P} \coloneqq \bigcup_{i=1}^{n} \{\text{Prov}^i_{\mathcal{A}}, \text{Prov}^i_{\mathcal{C}}\}$.

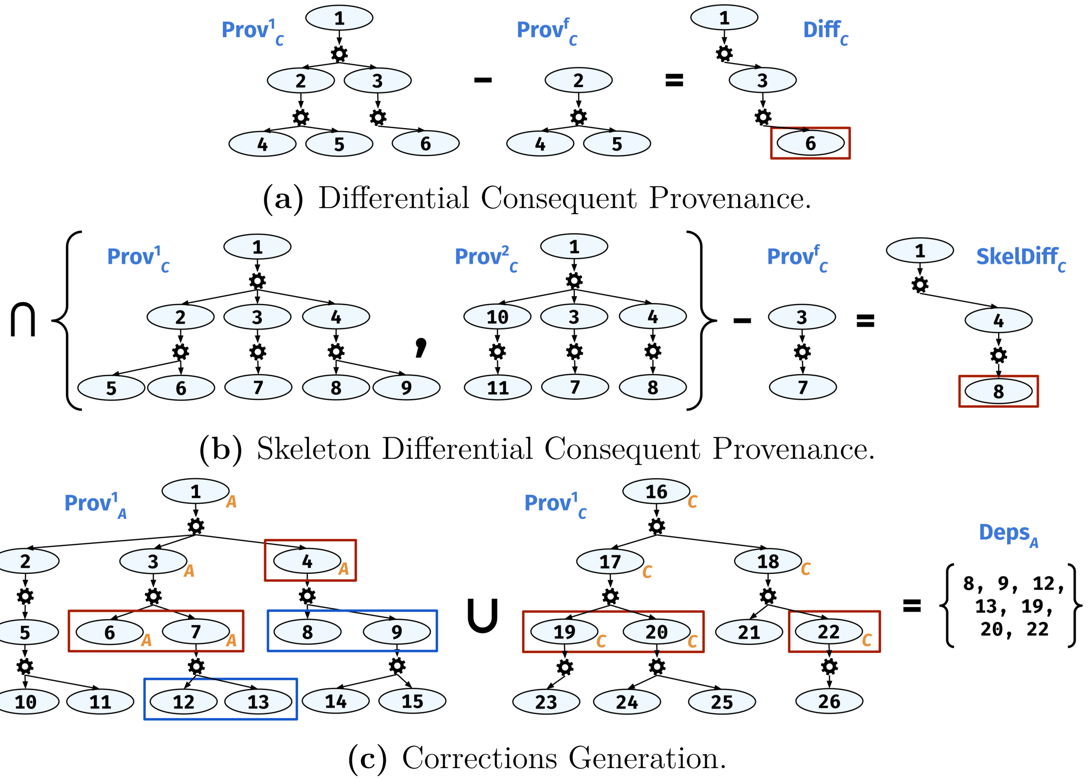
Independently, the provenance graphs are of little immediate use for distributed debugging, as we saw in Section 5.1. But as we will see, a variety of

simple *queries* over these graphs helps reveal both root causes of observed bugs as well as - surprisingly - potential bug fixes. To enable such queries we require a collection of graph operations, each of which produces a new graph when applied to elements of $\mathcal{P}$. For intuition, we pun on the set operations *intersection* ($\cap$), *union* ($\cup$), and *difference* ($-$). All graph operations work as we expect them to. $A \cap B$ produces the graph that only contains vertices and edges that A and B share. $A \cup B$ yields the graph with all vertices and edges from A or B or both. $A - B$ gives us what is left of A when all vertices and edges of B are removed. Intersection and union seamlessly work for more than two graphs at once.

We need to be able to select specific vertices from the provenance graphs in $\mathcal{P}$ and applications of the graph operations among its members. Thus, we briefly introduce a number of integral vertex selection functions informally. Function $\texttt{prop}_{x=y}(A)$ returns the subgraph of A for which property x equals y on all vertices. Function $\texttt{normalize}(A)$ yields the reduced and simplified standard form of provenance graph A, i.e., a more abstract representation of A where run specifics are hidden, e.g., by collapsing chains of the same event type into one, etc. Function $\texttt{leaves}(A)$ produces all vertices of A without any outgoing edge. Analogously, $\texttt{roots}(A)$ returns all vertices without any incoming edge. Considering a subset V of the vertices of graph A, $\texttt{reachable}_A(V)$ yields all vertices in A reachable from each element in V.

5.2.4 Principal Strategies

We now show how to use our framework to express common debugging strategies that expose causes of distributed bugs and assist developers in writing fixes.



(a) Differential Consequent Provenance.

(b) Skeleton Differential Consequent Provenance.

(c) Corrections Generation.

Figure 5.4: Exemplary visualization of our three principal strategies for provenance-based debugging. Per strategy, equal vertex numbers identify the same logical event. Red boxes denote the result of operation `leaves` on a subgraph, blue boxes show the outcome of operation `reachable`. Orange-colored indices mark that the respective property evaluates to true for that vertex.

Differential Consequent Provenance

Differential provenance [61] aids in root cause analysis by revealing a *frontier* - a line distinguishing the point at which the failed run departed from the successful path - highlighting events that failed to occur. Expressing differential provenance in our framework is straightforward. By construction, the first run is successful, i.e., for run 1 it holds that $\mathcal{A} \rightarrow \mathcal{C}$. Let run f be the failed run, i.e., $\mathcal{A}$ holds but $\mathcal{C}$ does not at test end. We can now reason about the set of program rules $\text{Diff}_{\mathcal{C}}$ that did not execute in the attempt of establishing $\mathcal{C}$ in the failed run, by issuing

the following query in our framework:

$$\text{Diff}_{\mathcal{C}} \coloneqq \texttt{leaves}(\text{Prov}^{1}_{\mathcal{C}} - \text{Prov}^{f}_{\mathcal{C}})$$

We visualize the resulting set of vertices $\text{Diff}_{\mathcal{C}}$ over abstract provenance graphs in Figure 5.4a. Changing the program to ensure that the statements in $\text{Diff}_{\mathcal{C}}$ always execute is sufficient to repair the bug, but how should the programmer do so? If the problem is an error of commission, the appropriate fix will often involve making a change to the program that is *near* the frontier identified in $\text{Diff}_{\mathcal{C}}$ - for example, by repairing an off-by-one error. Differential provenance can help debug some errors of omission as well. For example, if the bug involved an unhandled exception, the code that threw the exception is likely to be close to the unexecuted statements in $\text{Diff}_{\mathcal{C}}$, and hence the appropriate repair will be close as well.

Unfortunately, repairs for errors of omission are not always straightforward, and this approach can be a dead end. Consider again the protocol presented in Section 5.1. $\text{Diff}_{\mathcal{C}}$ identifies the rules that failed to fire when messages were dropped between the primary and backups. Focusing narrowly on this slice of the program, the obvious fix would appear to be retrying these messages in order to overcome loss. But for any pattern of retransmission, there is a corresponding pattern of loss, and an intelligent bug finder will find it! The fundamental flaw of the program is the primary acknowledges the client *too soon.* Differential provenance alone leads us away from this bug.

Skeleton Differential Consequent Provenance

When more than one successful run is available, we can take the idea of extensions based on differential provenance one step further. Instead of relying on only one successful run to determine what comprises success, we use all of them and

create a *skeleton* - essentially, the prototype of a successful run. Let $f \geq 3$ denote the failed run again. We thus have at least two successful runs available for our query. Let $s = f - 1$, such that $1, \ldots, s$ refer to the respective successful runs. Incorporating the idea of a "protocol core extraction", reduces to the task of intersecting the consequent provenance graphs of all successful runs prior to obtaining their difference set $\mathrm{SkelDiff}_{\mathcal{C}}$ with the failed run's consequent provenance:

$$\mathrm{SkelDiff}_{\mathcal{C}} \coloneqq \mathtt{leaves}((\bigcap_{i=1}^{s} \mathtt{normalize}(\mathrm{Prov}_{\mathcal{C}}^{i})) - \mathrm{Prov}_{\mathcal{C}}^{f})$$

For intuition, we show a simplified computation of the vertex set $\mathrm{SkelDiff}_{\mathcal{C}}$ in Figure 5.4b. Oftentimes, protocol runs vary slightly in flow, e.g., in specific number of message retries due to coping with message loss. By focusing on rules present in all successful runs, we aim to remove important but secondary protocol behavior. This helps us direct attention on increasing redundancy of indispensable yet missing program rules in the failed run. Trying to look beyond specific features of the individual successful runs, we suggest to introduce redundancy updates that enable the rules $\mathrm{SkelDiff}_{\mathcal{C}}$ to fire under more fault settings.

Corrections Generation

The two debugging strategies above provide us with high resolution pointers into program logic, guiding the programmer's attention to regions of the program where it is likely that the bug lies. But as we discussed in Section 5.2.4, for some classes of omission bugs there simply is no code region that requires repair; rather, as in the case of asynchronous primary/backup, the protocol has been insufficiently developed and additional program logic needs to be added.

We have one other tool at our disposal, however: the correctness specifications themselves. If we reconsider the structure of our correctness specification $\mathcal{A} \rightarrow \mathcal{C}$,

we know that when we observed a failed run, $\mathcal{A}$ held but $\mathcal{C}$ did not. Thus, there must exist a window in the protocol flow during which an injection of the right combination of omission faults will leave the protocol no chance to ever establish $\mathcal{C}$ before the test ends. While increasing the number of ways for $\mathcal{C}$ to eventually be established makes the protocol more robust, it only delays the time at which the bug finder injects the right omission faults that forfeit $\mathcal{C}$ once more.

Going back to our protocol from Figure 5.1 that is supposed to provide durable replication, we see that no matter how often we instruct the primary to send `replicate` messages again, dropping all of them or crashing all replicas will still be successful in preventing $\mathcal{C}$ from being established. No matter how many redundancy measures we add, an intelligent bug finder always comes back with at least one run that violates the specification. We need to switch tactics and make our protocol correct first, before the increased robustness becomes visible. Most of the resources for automatically generating, applying, and testing these protocol *corrections* are already available. Specifically, what needs to change are the conditions under which we consider $\mathcal{A}$ established. We need to make sure all conditions for establishing $\mathcal{C}$ become conditions for establishing $\mathcal{A}$ as well. We identify these rules triggering $\mathcal{C}$ and generate updated dependencies for $\mathcal{A}$ that precisely include those that cause $\mathcal{C}$ to be true. Put differently, only report a good protocol state being achieved ($\mathcal{A}$), when we know the consequent state ($\mathcal{C}$) has already been as well. We obtain the updated dependencies set $\text{Deps}_{\mathcal{A}}$ by querying:

$$\begin{aligned}\text{Deps}_{\mathcal{A}} \coloneqq\ & \texttt{reachable}_{\text{Prov}^1_{\mathcal{A}}}(\texttt{leaves}(\texttt{prop}_{\mathcal{A}=true}(\text{Prov}^1_{\mathcal{A}}))) \\ & \cup\ \texttt{leaves}(\texttt{prop}_{\mathcal{C}=true}(\text{Prov}^1_{\mathcal{C}}))\end{aligned}$$

Omitting details of an actual protocol execution, the updated dependencies set $\text{Deps}_{\mathcal{A}}$ for $\mathcal{A}$ based on exemplary provenance graphs $\text{Prov}^1_{\mathcal{A}}$ and $\text{Prov}^1_{\mathcal{C}}$ is shown in

Figure 5.4c. Distributed specifications such as durability naturally take a global view on system state dispersed across the members when verifying $\mathcal{A}$ and $\mathcal{C}$. In case verifying $\mathcal{C}$ in a buggy protocol indeed ranges over more than one node, it does not suffice to simply add the missing triggers for $\mathcal{C}$ as dependencies to $\mathcal{A}$, due to their separate logical locations. Instead, communication schemes are required that allow all nodes establishing $\mathcal{A}$ to reason about remote state on all nodes establishing $\mathcal{C}$. In these situations, $\text{Deps}_{\mathcal{A}}$ will differ such that $\texttt{leaves}(\texttt{prop}_{\mathcal{C}=true}(\text{Prov}^1_{\mathcal{C}}))$ is replaced with knowledge about the remote states through messages.

Invoking this strategy, the programmer will be presented with a set of rule suggestions to add and a set of dependencies to adjust, that, if applied appropriately, close the window between establishment of $\mathcal{A}$ and $\mathcal{C}$ permanently fixing the bug. While final adjustments have to be made by the programmer, we will see in Section 5.3 that these appear easy enough for developers inexperienced with the protocol to devise and insert into the protocol code.

5.3 Evaluation

We validate our debugging strategies using real-world bugs from the TaxDC collection by Leesatapornwongsa et al. [60]. The collection describes, labels, and categorizes distributed concurrency bugs, i.e., bugs caused by the non-determinism of distributed events inherent to distributed systems. Based on bug tracker reports from large-scale distributed systems such as Cassandra, Hadoop MapReduce, HBase, and ZooKeeper, Leesatapornwongsa et al. extract triggering conditions, a succinct description of steps leading to the bug, and official fix if available. We reviewed 52 of the available 104 TaxDC bugs, chosen arbitrarily after a rudimentary screening, which we classified according to root cause, noting for each class whether it is correctable or debuggable with our framework. We present the

resulting taxonomy in Section 5.3.1 and Table 5.1. We implemented the principal strategies from Section 5.2.4 in our prototype debugger Nemo [59] and successfully analyzed and fixed a subset of the bugs from our taxonomy. We present four case studies to demonstrate effectiveness and limitations of Nemo in Section 5.3.2.

Table 5.1: Taxonomy of 52 distributed concurrency bugs from the TaxDC collection and the asynchronous primary/backup protocol from Figure 5.1. Legend: ✔ = yes, ✘ = no, ❍ = it depends.

	Bug Class & Description	Corrections	Assistance	Bugs	Canonical Fix
Timing	*message-message* Two messages race each other.	✔	✔	9	*Sending node checks specification:* Add communication about progress of local event before sending message.
	message-local A message races a local event.	✔	✔	14	*Local node checks specification:* Add message queue between sender and node. Wait for message delivery or computation completion before progressing.
	local-local Two local events race each other.	✔	✔	1	

Continued on next page

Table 5.1 – continued from previous page

	Bug Class & Description	Correc-tions	Assis-tance	Bugs	Canonical Fix
	premature success Consequent races with end of test.	✔	✔	Async P/B	Add communication about consequent state in system to nodes enforcing specification. Expand success conditions by positive response.
Logic	*state transition* State transition in response to an event is wrong.	✘	✔	11	*Fix:* Add missing transition for unexpected event. *Assistance:* Differential provenance points to missing completion event of vulnerable state.
	config Misconfiguration.	✘	✔	1	*Fix:* Configure system correctly. *Assistance:* Differential provenance points to goal that differs in specific config value.
	fallback behavior Actions in response to perceived errors are wrong.	✘	✘	5	*Fix:* Rewrite or add wrong fallback logic. *Assistance:* None.

Continued on next page

Table 5.1 – continued from previous page

Bug Class & Description	Corrections	Assistance	Bugs	Canonical Fix
bug Concept or implementation error.	✘	❍	11	*Fix:* Depends on bug. *Assistance:* Depends on bug.

5.3.1 Bug Taxonomy

In Table 5.1, we categorize distributed concurrency bugs into bugs due to *timing* issues and bugs due to node-local *logic* mistakes. These root causes correspond almost precisely with our informal rubric of omission (timing) and commission (logic) errors. Prominent representatives for the first category are race conditions. We distinguish *message-message*, *message-local*, and *local-local* races, where *message* is a data item in network transit and *local* a node-local computation. As the TaxDC bugs do not come with a correctness specification of the form $\mathcal{A} \rightarrow \mathcal{C}$, most races come down to event order on one node. Thus, category *premature success* for bugs where $\mathcal{A}$ is established too permissively and $\mathcal{C}$ fails to be established until test end due omission faults, currently only holds for our protocol from Figure 5.1. On the other end of the spectrum, root causes of logic bugs ultimately amount to node-local logic errors. Bugs of this type continue to occur even when all omission faults have been incapacitated. We classify further into bugs in which a protocol stops working correctly due to a wrong or missing *state transition* in response to an event, has been run with a wrong *configuration*, does not have any or the wrong *fallback behavior* to errors, or features an implementation *bug*.

Of 52 bugs, 24 are potentially repairable by our corrections generation strategy (Table 5.1, column "Corrections"). These are precisely the bugs in the timing category, demonstrating the ability of our framework to help fix these errors of omission. The remaining 28 bugs root in logic mistakes and thus cannot be corrected through generated protocol-level changes. However, debugging 12 of them will reduce to highly-targeted rule comparisons by assistance of our queries rooted in differential provenance (Table 5.1, column "Assistance"). Further 11 bugs are general mistakes and the effectiveness of our methods highly depends on the bug at hand. Finally, only for bugs with wrong fallback behavior, our strategies provide no advantage in assistance over conventional debugging methods.

5.3.2 Case Studies

We implemented three timing and three logic bugs from Table 5.1 in Dedalus [66] and submitted them to Molly [69], the reference implementation of lineage-driven fault injection [15]. For each, Molly found omission faults violating their correctness specification. We confirmed the effectiveness of our corrections strategy by successfully fixing the timing bugs—we present how so below. Additionally, we show how Nemo brings us in close proximity of the root cause when analyzing one of the logic bugs it cannot automatically repair.

CA-2083 (Message-Message Race). We start with Cassandra bug 2083, representative of the class message-message races in which protocols behave correctly when messages are received in expected order, but violate their specification in the event of a network reordering. In CA-2083, a schema message creating a new keyspace and a data message carrying data for the new schema race to one of the nodes. If the data message unexpectedly arrives first, it will get dropped because of the unknown keyspace. The canonical and official fix is to buffer the

data message if it is received first and enforce processing of schema message prior to delivering the data message. Nemo identifies this race and synthesizes a modification of one line of protocol code that results in enforcement of the correct order. A subsequent Molly-Nemo loop confirms our success. Additionally, Nemo suggests improving the fault tolerance of some critical network events prone to omissions. When included, we obtain a correct protocol resistant to severe message loss.

ZK-1270 (Message-Local Race). ZooKeeper bug 1270 is a race not between messages but a message and a local computation that runs for longer than expected. After an election, a new leader sends a confirmation message to a follower and awaits a response, which it can only accept after moving to AWAIT state. If this computation is delayed (e.g., due to a garbage collection pause), the leader could receive a response before transitioning, and ignore the reply. When it eventually moves to AWAIT, it blocks, because it will never receive another message. The official fix delays response delivery until the transition completed. Nemo resolves the race by synthesizing a single line of code enforcing the ordering constraint: `success(L) :- sent_flag(L), ack(F)`. Here, adding `sent_flag(L)` to the dependencies for leader `L` to ultimately declare a run a success prevents a run from prematurely becoming successful in case an acknowledgment is processed before the leader moved to AWAIT. After repair is confirmed, Nemo suggests improvements in the form of end-to-end retries of confirmation messages.

MR-2995 (State Transition). In Hadoop MapReduce bug 2995, we face a local-logic state transition bug. A manager is prone to crash when it receives an expiration instruction for a resource it is still initializing. No protocol-level change that Nemo can generate will fix this root cause. Nemo falls back to differential provenance in this case, identifying the first program statement that fired in the successful execution but failed to fire in the faulty one: the "completion" message

indicating that initialization succeeded. The programmer will need to rewrite this line of code, to either ignore the expiration message or delay its processing.

Async P/B (Premature Success). We close the circle by returning to our protocol from Figure 5.1. Due to premature optimizations, a client considers its payload durable as soon as it has received acknowledgment from the primary, but before verification of payload presence in all node logs. We reason about global system state when verifying the specification, which distinguishes Async P/B from above race conditions. The fix is to ensure the client knows its payload to be durable before declaring success. Nemo suggests to introduce `ack_log` to inform the client about replica state and making receipt of `ack_log` from all nodes condition for success. All in all, Nemo proposes to modify four lines of code, after which a subsequent run confirms our success in eliminating the bug and indeed making the system durable. Additional fault tolerance analysis suggests to increase the resilience of rules `replicate`, `request`, `ack_log`, and `ack`, leading to a correct and more robust primary/backup replication protocol that resembles in code what the specification describes as correct.

Via Nemo, we showed strong evidence that the question-and-answer process of bug identification and repair can be posed as queries over traces of system executions, identifying causes of errors of commission. We also demonstrated Nemo's surprising ability to use this provenance querying framework to synthesize protocol repairs which cause the program to more closely fit its specification in the case of errors of omission. Nemo operates on an idealized model in which distributed executions are centrally simulated, record-level provenance of these executions is automatically collected, and computer readable correctness specifications are available. In the next chapter, we discuss incident localization for large-scale distributed systems with shallow or non-existent specifications, coarse-grained trac-

ing and logging rather than provenance collection and detail the challenges we needed to overcome to achieve efficient and effective incident localization.

Chapter 6

Troubleshooting: Incident Localization

In this chapter, we consider troubleshooting in production systems. Many large production systems are composed of thousands of microservices with instances in several regions and availability zones. Systems constantly evolve as new functionality is developed, bug fixes are pushed out and old services are phased out, to name a few examples. When a system does not behave correctly for some fraction of users, Site Reliability Engineers (SREs) declare incidents. Incidents are common but downtime is expensive. Outages of even a few minutes can cost service providers hundreds of thousands of dollars in revenue [70, 71]. Additionally, they incur soft costs in terms of poor user experience. The priority for SREs is to restore system functionality quickly. Before they can act to mitigate system unavailability, SREs must first *localize* the incident. Incident localization is the process of identifying a location - a component (hardware or software) - where a mitigating action may be applied. For example, if SREs determine that the behavior of service instances in a particular data center is problematic, the action recommended by SREs might be to divert traffic away from it. Other examples of

mitigating actions include re-configuring access control or firewall rules, reverting a code change and restarting components.

Incident localization is difficult in practice for two reasons. First, since distributed applications are complex and highly connected, SREs need to consider large volumes of data from varied sources (metrics, logs, events, and traces) generated by executions before and during the incident to reason about system behavior. Based on their observations, SREs then attempt to determine a pattern in *how* executions fail during an incident. Second, many events in the failing executions may be different from the successful executions. SREs have to infer the relationships between different events for effective (correct and precise) localization; a time-consuming process. SREs have access to a suite of tools but may need to use multiple tools to obtain insights from different data sources. Outputs from one tool may be modified and used as inputs to a different tool.

Since traces capture events within a user request and how they relate to each other, the relationships captured in traces are precisely those which SREs currently infer manually, recommending the use of traces. Combining differential reasoning with comparing sets of traces helps determine the consistent structural changes across traces (change pattern) during an incident. Thus, using traces as a data source addresses both causes of slow incident localization.

The key idea of our approach, Aggregate Comparison of Traces (ACT), is to find events present in one set of traces but not the other and then use the structure of individual traces to reason about cause and effect. ACT leverages the relationships captured between events within traces as opposed to SREs manually connecting the dots. Thus, we are able to focus SREs attention on a few (ideally one) events or relationships that they need to investigate further to recommend an effective mitigating action.

We evaluate ACT on datasets from HDFS [72], DeathStarBench [73], and eBay. In our quantitative experiments, we conduct hundreds of simulations for three different failure modes and show that ACT is able to identify a mitigation site that enables effective action in all but a handful of cases as compared with our baselines that produce irrelevant results in 30-50% of the cases. For SREs mitigating incidents, the above result implies that ACT identifies exactly where the mitigation is to be applied. We have integrated ACT with Jaeger [34], an open source tracing tool, for online trace comparison. ACT opens a line of inquiry into using groups of traces for incident localization, which, if adopted widely, can change the way SREs approach incident response. We also contrast ACT with approaches taken by commercial tools such as Lightstep [35].

The rest of the chapter is organized as follows: In Section 6.1, we first present details of an incident study which has two main takeaways. First, we provide evidence to show that incident localization dominates response time. Second, we show that RPC failures produce the largest impact to system availability. Therefore, we focus on incidents that arise from RPC failures in our work. With an example incident, we motivate the use of traces to localize incidents by discussing relevant approaches from the state of the art and highlighting their shortcomings. Section 6.2 develops an approach that compares sets of traces and analyzes their events and relationships to localize incidents. In Section 6.3, we focus on evaluating ACT vis-a-vis baselines that adapt approaches from prior work to our setting and finally, we touch upon the details of integrating ACT with Jaeger. We close with Section 6.4 in which we motivate the need for iterative localization, describe *how* we extend ACT to drill down to specifics and present some preliminary results from applying these on executions of sample open-source applications [51].

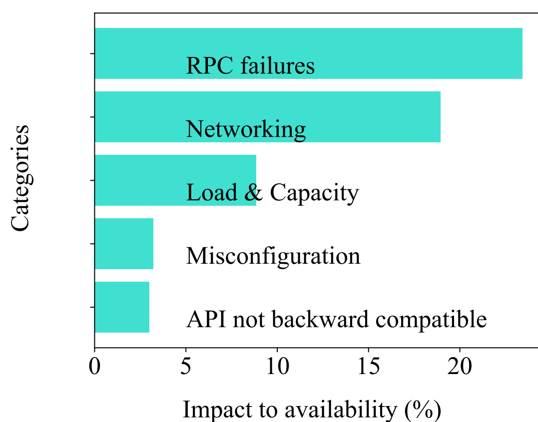


Figure 6.1: Percentage of impact by category - we have represented the top five of over a dozen different categories that emerged based on available data. Incidents arising due to breakdown in communication between components at the application level have the highest impact.

6.1 Background and Motivation

The principal goal of incident mitigation is to minimize impact to users. Understanding the causes of the incident is usually a secondary goal, often a more costly (in terms of time and effort) exercise reserved for post-incident reviews.

SREs use aggregate alerts from metrics deviations, logs from services, etc to build a mental model of the system. These models are often based on tribal knowledge, typically incomplete and usually outdated [74–76]. We first present observations from a study of incidents at eBay.

6.1.1 Incident Study

To determine trends in incidents, we studied incident reports of 75+ severe incidents that occurred over three years (June 2017 - May 2020) at eBay. Severe incidents correspond to more than 85% of overall impact. Incident impact is measured in terms of loss of availability. Impact to availability of an incident is the duration of time that all or some fraction of users were unable to use the

system i.e. availability of service. We make the following observations:

Highest impact is from RPC failures between components at the application level: Figure 6.1 shows the top five incident categories in order of decreasing impact. We have not represented incidents arising from vendor issues since reading incident reports only gives us a partial view of these incidents. RPC failures between components includes:

Component down: Components can fail for various reasons - a recent change made to the component, a dormant bug in a code path not used often - triggered by an increase in load or a change in user options. Component failure by itself is not an incident but the lack of a fallback mechanism or the critical nature of a component not being common knowledge can lead to an incident. Incidents corresponding to this category may arise due to component failures or decommissioning components that are in use.

Component A unable to call component B: A component (A) which was previously able to make RPC calls to a different component (B) that it depends on may no longer be able to do so due to link failures, changes in access control lists, firewall rules, and security fixes. This may further result in additional, unexpected component interactions.

Buggy failure recovery: When a component fails to respond within the configured timeout or crashes, the calling component may call a different component or perform a series of actions to recover. Since the recovery path is generally not exercised often, it may not work as expected resulting in overall request failure. In such a case, the fallback may be inappropriate or incorrectly set up to recover from the failure. This is an important failure mode as failures in the recovery path continue to be reported [77–79] despite various efforts to address them.

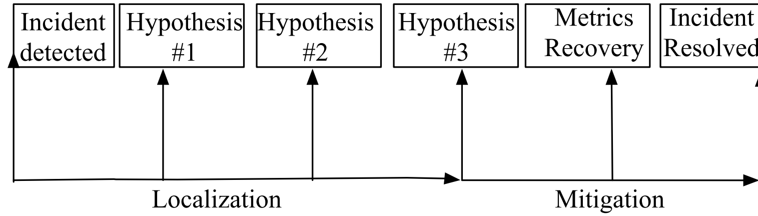


Figure 6.2: Typical incident timeline

About half the incidents were localized incorrectly at least once: Figure 6.2 shows a simplified timeline from incident detection to resolution. Localization time and mitigation time are the times taken to effectively localize an incident and apply the mitigating actions that effect recovery respectively. Incorrect localizations (Hypothesis#1 and Hypothesis#2) for an incident indicate that there were one or more mitigation steps that were pursued before the incident was effectively localized. A mitigating action that does not result in metrics recovery prolongs poor user experience and increases revenue impact. Prior works [80–82] indicate that most incidents are reassigned at least once during triage and that triage time dominates response time [82]. Reassigning incidents results in a longer time to apply a mitigating action and therefore, slower incident response.

Takeaway: Effective localization of incidents that arise from RPC failures between application-level components would produce the most significant reduction in user impact and therefore, we focus on these.

6.1.2 Motivating Example

We describe a real incident that occurred at eBay which serves as our running example for the remainder of the section. Figure 6.3 depicts the actions taken by SREs. The mitigation steps took SREs close to 3 hours and was dominated by time taken to arrive at the correct mitigating action (2.5 hours).

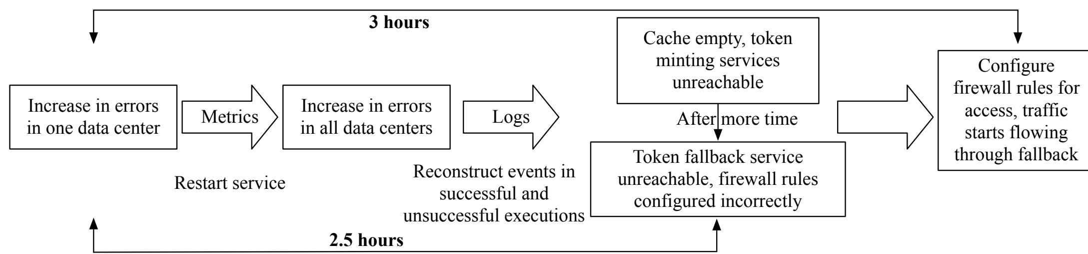


Figure 6.3: This figure represents how SREs responded to an incident and the data sources they used (logs and metrics). The mitigation steps took SREs close to three hours, two and half of which was arriving at the correct mitigating action.

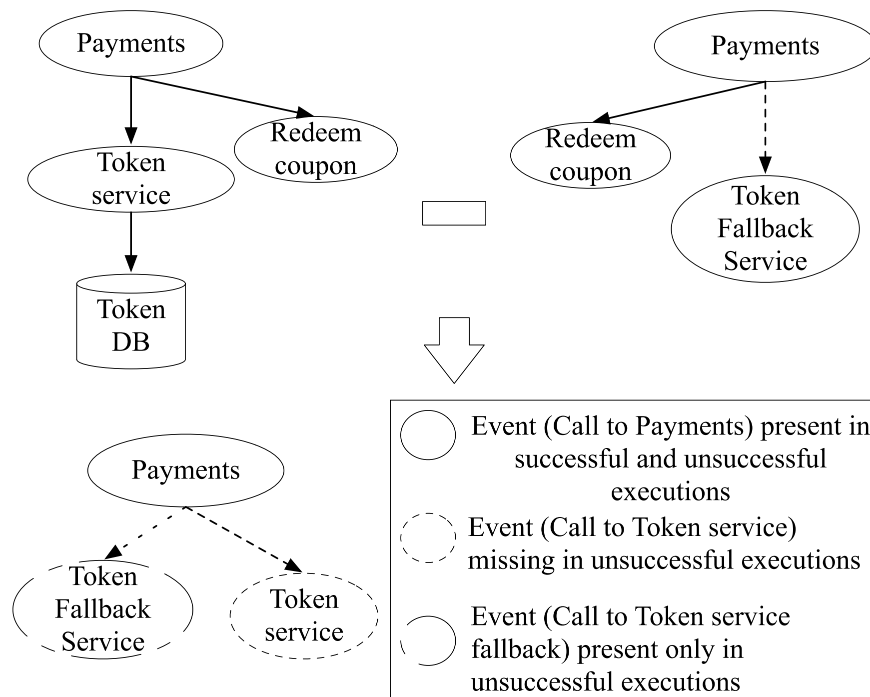


Figure 6.4: This is an idealized picture of graph differencing and contains only the relevant services. On the left is a partial view of a successful request where the token service was working as expected. On the right, we have the trace, after the restart of the payments service which continued to see errors due to incorrect firewall rules.

SREs first observed an increase in errors (a metric) for the *Payments Service* in one data center and immediately declared an incident. Metrics are used to monitor the overall health of the system. Business metrics such as number of transactions completed, number of canceled transactions and rate of incoming

traffic are tracked in real-time since they are related to revenue. SREs attempted to mitigate the incident by restarting the service, but errors increased in all the data centers instead. In this instance, SREs could surmise from the metrics *that* something was wrong, but not what or why.

To understand what caused the increase in errors, SREs looked at the application logs and noticed that the local cache used for storing access tokens was empty and the service used for minting the tokens (*Token Service*) was unreachable. SREs found that *Payments Service* started calling *Token Fallback Service* instead of *Token Service*. However, all calls to *Token Fallback Service* were also failing. Further investigation using the logs revealed that the *Token Fallback Service* was inaccessible due to incorrect firewall rules. Once the firewall rules were corrected, the error rate returned to normal and *Payments Service* fully recovered.

The breakthrough in our running example came when one of the SREs observed *from the logs* that during the incident, requests were attempting to make a call to a service (*Token Fallback Service*). No such call was present in pre-incident execution traces. SREs had to trawl through logs to find the specific events and event interactions in the unsuccessful executions that contributed to its failure. These events indicated the presence of *additional* calls that were not present in executions before the incident. In this case, SREs needed to not only understand the *absence* of calls from the logs but also the *presence* of additional calls. This illustrates that effectively localizing incidents usually requires both aggregate (the presence of errors) and causal information (*Payments Service* trying to call *Token Fallback Service*) - in this instance provided by metrics and logs. SREs had to first determine which executions to consider based on the failure of requests and then compare the events and their relationships between successful and unsuccessful executions. The request path and system model were reconstructed from system

logs for this incident. The crucial step in localizing the incident was differencing the request paths before and during the incident to see what was different about the request paths.

6.1.3 Limitations of existing approaches

Although prior work localizing incidents in data centers [83, 84] is extensive, these are orthogonal to localizing incidents at the application-level since applications are designed to tolerate network failures such as link failures and packet drops. For example, a service usually has instances in multiple data centers such that if a network link in one data center goes out, requests will be sent to a different instance.

We will briefly describe each of the observability signals - metrics, logs, and traces - and the most relevant approaches that use them as inputs. With our running example as context, we discuss why they don't effectively localize incidents. We also discuss the constraints of differential reasoning when using traces and how our approach addresses them.

Fa [1] detects and localizes incidents by vectorizing metrics to learn incident signatures. The localization points to the set of metrics (and underlying components) most relevant to the failure. During an incident, multiple metrics are affected and SREs would need to understand relationships amongst different components. Marianil et al. [2] use metrics to learn a baseline model and build out an undirected graph by correlating pairs of metrics. They further use graph centrality measures to identify the most severely affected metrics and thereby, a faulty service. For our example incident, the Payments Service has the most errors, and will most likely be identified as the faulty service. This does not give SREs any actionable insights and is therefore, not useful.

More recent approaches such as Grano [4] and Groot [5] assume that relationships amongst components are available either in the form of system architecture diagrams or global dependency graphs. They build machine learning models to identify the metrics correlated with a given incident which are then overlaid on the dependency graph for incident localization. Such follow-the-errors approaches do not work when multiple incidents co-occur, one or more metrics are not captured or incident localization involves identifying when a call between two components *did not occur*. Our example incident falls into this last category.

Logs capture a machine centric view of the system and provide additional context, but require sifting through large volumes of data to extract it. Aggarwal et al. [3] model logs from different components as multiple time series and correlate errors emitted by various services to localize the incident given a dependency graph (static topology or architecture diagram). Approaches that reconstruct individual user requests from logs involve control and data flow analysis [10, 12]. Yet others use unique identifiers to identify events corresponding to different requests [9] and custom log parsing to recognize identifiers [11]. Network communication and temporal order are used as heuristics to infer relationship amongst events. Causality inference using log analysis is brittle since it depends on the quality of user logging and is inapplicable either due to practical concerns (running control and data flow analysis for constantly evolving systems like microservices with continuously changing topologies is impractical) or because the timescales for incident localization are very stringent (as systems scale, application logs grow, increasing analysis time).

Distributed tracing provides a request-level view of the system and is used for debugging [6, 7, 16], profiling, and monitoring production applications. It has also been used to address correctness concerns [28], for capacity planning, and

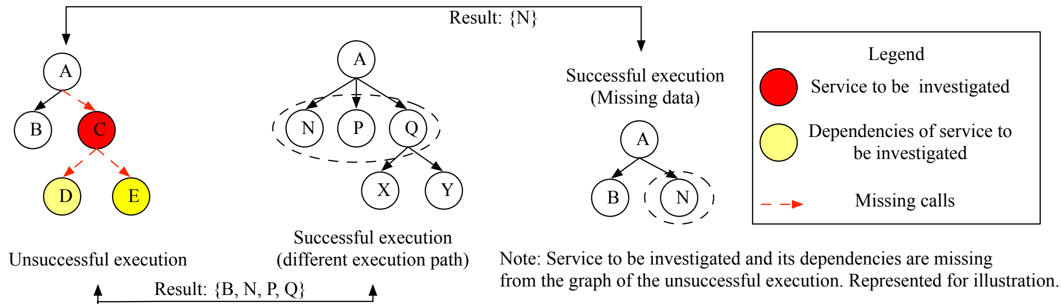


Figure 6.5: Limitations of pairwise comparison - the two examples demonstrate the circumstances when pairwise comparison produces false alarms and can occur either separately or in combination.

workload modeling [8]. Tracing is increasingly being adopted by industry and there is a push for standardization [33, 85, 86] as well. A trace captures events that occur in a given request as well as how they relate to each other i.e causality. The most general representation of a trace is a directed acyclic graph (DAG) where nodes and edges correspond to events and their interactions respectively.

Had traces been available, SREs would been able to compare the trace of a successful execution and that of an unsuccessful execution - which we call *pairwise comparison* - to determine the events that differentiate the two. Doing so would have highlighted the missing and additional events in executions during the incident and thereby enabled them to take effective action. Figure 6.4 demonstrates an idealized result of pairwise comparison for our running example. In the unsuccessful execution, the call from *Payments Service* to *Token Service* service is missing, but an attempted call from *Payments Service* to *Token Fallback Service* is additional. Since the structure of a trace represents causality of event interactions, we use it to establish cause-and-effect relationships between events in the result - retaining only the causes.

Prior works that uses trace analysis employ a similar differential approach between pairs of traces with appropriate user inputs. For eg., ShiViz [6] and

Jaeger [34] both support pairwise comparison of user selected traces. Such tools allow SREs to interactively validate hypotheses but are not suited to automated incident localization.

Pairwise Comparison: A deep dive

The most important requirement for automated localization using pairwise comparison is *graph selection* - selecting a successful and an unsuccessful execution that exercise the same code path. In large, distributed systems, requests with identical inputs can often take different paths due to cache effects, dynamic request routing, traffic shifting across data centers, experimentation, etc. Traces generated from such requests may have different structures wholly or partially. Further, the structure of traces can also change with configuration changes in the application and deployment environment, ongoing code deployments, new feature deployment and code deprecation. At any given time, several such changes to request paths exist in production. Comparing pairs of graphs corresponding to different executions paths will produce incorrect localizations.

Further exacerbating the problem of graph selection is the fact that tracing is best-effort. That is, for some executions, the trace corresponding to the execution may be missing some data. Figure 6.5 demonstrates incorrect localizations produced by comparing executions that exercise different execution paths and when comparing incomplete traces of similar executions. Therefore, choosing a pair of graphs to compare based only on their structure is not viable.

Prior work makes simplifying assumptions about the system under consideration to make graph selection viable. Magpie [87] assumes a static system model and learns a probabilistic model of the system. An unsuccessful execution would deviate from the model and the difference between two such traces represents the

localization. Large distributed systems (open source systems eg., HDFS, HBase and commercial systems eg., Netflix, AWS) are complex and constantly evolving, invalidating this assumption. Spectroscope [7] assumes that a small number of unique execution paths exist in the system compared with the large number of underlying system traces, an assumption that does not hold for any but the smallest systems. GMTA [88] makes two assumptions. First, it assumes that the model is known and traces can be accurately labelled based on the functionality they exercise. Second, it assumes that for each label, there exists a single canonical graph. The first assumption requires that the labelling be kept up-to-date with changing models and the second assumption only holds if there exists only a single execution path for given functionality, a premise that is untrue for large systems, as discussed at the beginning of the section. In summary, the simplifying assumptions made do not hold for large distributed systems.

In our work, we sidestep the problem of graph selection by considering *sets* of traces rather than selecting a single pair of traces. From these, we derive *aggregate* insights while preserving useful information for difference based diagnosis. Lightstep [35] represents the closest industry tool to ACT and addresses some of the same failure modes. Lightstep also compares traces in aggregate, but focuses on finding tags or markers in the traces containing errors. However, for failures in the recovery path, being able to identify that a call was not successful does not help with determining a mitigating action. In our example, Lightstep would follow the errors to the failed call from *Payments service* to *Token service.* This only represents one half of the localization and the incident could only be effectively mitigated by SREs understanding that the call to *Token Fallback service* also failed in an attempt to recover from the failed call to *Token service* - i.e. knowledge of both the missing call and the additional call. Furthermore, success

or failure is an end to end property of a request and typically cannot be derived from a trace. For eg., a service returning an error in a user request does not necessarily imply a failed request; rather, it may be an indication to re-try it at a later time.

In Section 6.2, we describe our approach that compares sets of traces from steady-state operation and during the incident and analyzes their events and interactions to localize incidents.

6.2 Design & Methodology

In our work, we use traces to localize incidents and thereby, speed up incident response. Localizing an incident highlights the absence (or presence) of event interactions during an incident that helps identify a location to apply a mitigation. Pairwise comparison is usually ineffective for localizing incidents since it produces false alarms, as described in Section 6.1. As we now show, we can precisely localize incidents by comparing *sets* of traces and using the *structure* of traces to separate effects from their potential causes, retaining only the causes.

We first describe View of a Trace, which enables trace comparison. It is not possible to directly compare traces since individual traces include details such as timestamps that are different for every trace and IP addresses that are not necessarily consistent between any two traces. By dropping attributes that are not consistent across traces, views of traces make traces comparable. For example, to debug issues when a service is unable to talk to another, retaining service names is sufficient. If, instead, we would like to debug issues that impact a subset of service instances, retaining service instance names when generating a view of a given trace can be helpful. In this work, we only retain component names when generating views. We denote views by View(T) and refer to elements of a view as

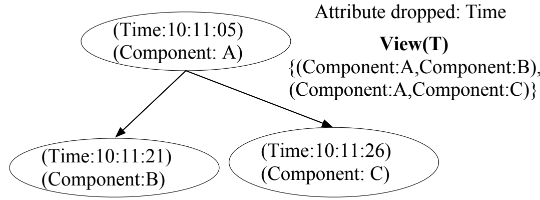


Figure 6.6: Simple trace and an example of a view

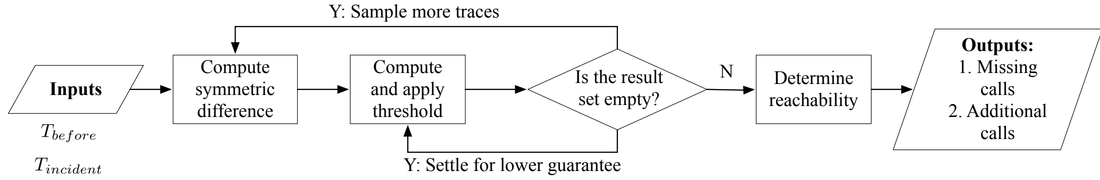


Figure 6.7: ACT consists of applying three techniques: Symmetric difference, thresholding and reachability - in that order.

ordered pairs. In Figure 6.6, for example, the ordered pair (Component:A, Component:B) in View(T) corresponds to the edge ((Time:10:11:05, Component:A), (Time:10:11:21, Component:B)) in the trace, T.

6.2.1 Inputs and Outputs

Inputs to ACT consist of two sets of traces - traces drawn during the most recent steady-state operation of the system (t_{before}) and traces drawn during the incident ($t_{incident}$). These are sampled based on the incident start time specified by SREs. We expect the sampled traces to satisfy two constraints. First, the number of traces sampled in each of the two sets must be large enough that a majority of events or event interactions, if captured in underlying traces, are present in the sampled traces. Many large-scale systems generate millions [31] of traces per day, but a much smaller sample size turns out to be sufficient for localization, as we will see in Section 6.3.1.

Second, traces are labeled as successful or unsuccessful based on an external

success criterion. Examples of external criteria could include credit card charged in case of buying an item, the item displayed correctly when it is added to the product catalog, a HTTP status code of 200, an acknowledgement of data writes, etc. If a trace cannot be assigned a label, it is discarded (less than 0.2% of traces).

Outputs from our system should localize the incident under consideration rather than return the entire difference between the set of traces before and during the incident. SREs can investigate along two axes - a) Why are specific calls missing during the incident? and/or b) Why are other calls present *only* during the incident? Based on what the investigation reveals, an appropriate mitigating action can be applied.

6.2.2 System Overview

In ACT, we use aggregate information from witnessing a large set of traces and the causality of event interactions within individual requests to localize incidents. Figure 6.7 shows the three techniques we use to localize an incident given sets of traces from before and during the incident.

The symmetric difference of t_{before} and $t_{incident}$ is the set of ordered pairs that are in one of $\bigcup_{i=1}^{|t_{before}|} View(Trace_i)$ or $\bigcup_{j=1}^{|t_{incident}|} View(Trace_j)$ but not both. If the changes produced by an incident are represented in $t_{incident}$ and at least one example of the correct interaction is in t_{before}, the symmetric difference *will* contain the site where the mitigation is to be applied. To obtain a precise result, we employ thresholding and reachability.

We use thresholding to answer the question: Which ordered pairs in the symmetric difference are statistically significant and must be retained? Since t_{before} and $t_{incident}$ are randomly sampled, one or more of the sampled traces may correspond to a code path that is rarely exercised. If such traces occur in one or the

other set of traces, some ordered pairs will be part of the symmetric difference as a result of sampling randomness. The use of thresholding allows us to discard these. Using a threshold also addresses trace quality issues in individual traces that arise due to the best-effort nature of tracing.

After computing symmetric difference and applying the threshold, the result may still contain some superfluous ordered pairs. To understand how this may occur, assume two ordered pairs (a, b) and (b, c) are in the result. The edges in a trace represent event interactions. For a given trace, let's further assume that (a, b) and (b, c) correspond to edges (e_1, e_2) and (e_2, e_3) respectively. Reachability is the transitive closure of the edge relation of a graph. If we find that (e_2, e_3) is reachable from (e_1, e_2), we can discard the ordered pair (b, c) since its potential cause (a, b) is in the result. By establishing cause-and-effect relationships between edges corresponding to ordered pairs and eliminating the ordered pairs corresponding to effects, we use reachability to whittle down the result set for effective localization. Failure of a database call or third party vendor issues into which SREs have no visibility can be localized to a single leaf node or edge. For these, we expect to see effective localization even without the use of reachability.

The three techniques build on each other - symmetric difference produces the initial result set while thresholding and reachability prune the result set such that the incident is effectively localized.

Techniques:

We describe in detail each of the techniques introduced in the previous section.

Symmetric Difference: To compute the symmetric difference, we only consider the successful executions in steady-state operation (unsuccessful requests in steady state could result from invalid credit card entry, insufficient stock, etc). t_s

represents successful executions in t_{before} and we shorten $t_{incident}$ to t_{inc} here. The result is the *entire* set of changes between the two sets of traces. If calls made in traces of successful executions during the incident are in the symmetric difference, these could not possibly have been caused by the incident. Therefore, we remove them from our symmetric difference. Let t_{inc_s} represent successful executions in t_{inc}. We obtain the symmetric difference as follows:

$$\begin{aligned}
Missing\ Calls\ (M) &= \bigcup_{i=1}^{|t_s|} View(t_{s(i)}) - \bigcup_{j=1}^{|t_{inc}|} View(t_{inc(j)}) \\
Additional\ Calls\ (A) &= \bigcup_{j=1}^{|t_{inc}|} View(t_{inc(j)}) - \bigcup_{i=1}^{|t_s|} View(t_{s(i)}) \\
&- \bigcup_{k=1}^{|t_{inc_s}|} View(t_{inc_s(k)}) \\
D &= M \bigcup A
\end{aligned}$$

Thresholding: We cannot use a flat threshold to determine the statistically significant ordered pairs because our threshold value can change not only as a result of system evolution but also based on the number of traces sampled. We derive our threshold as a function of frequency of calls in traces and the number of traces sampled. Frequency statistics can be computed in real time as traces are generated. Computed statistics can be stored in-memory since their memory footprint is small (order of a few hundred keys in a hash map).

$$\text{Threshold, } t \ = \ N \ * (1 - e^{log(0.01)/n})$$

We obtain t by solving for $(1-p)^n < 0.01$, where p is the probability that an ordered pair, c, occurs in at least t traces. The size of the corpus from which frequency statistics are computed is N and the number of sampled traces is n. Therefore, $p = \frac{t}{N}$. The threshold, t, is such that if a call appears in more than t

of N traces, there is a 99% probability that at least one trace containing the call will be present in a sample of n traces.

Given a threshold, if a call appears in more traces than the threshold and is unrelated to the incident, it will appear in both sets of traces with high probability and therefore not appear in the result. On the other hand, if the call is missing as a result of changes produced by an incident, evidence of the change will be seen in the sampled traces. Conversely, if the number of traces that a call occurs in is less than the threshold, it is discarded. SREs can choose a lower probability and re-compute the threshold for a less stringent guarantee.

Reachability: As discussed in Section 6.2.2, we exploit reachability to achieve the minimal result set. To do so, we use the structure of individual traces. Given two ordered pairs and a trace, T, we first determine the possible edges that each ordered pair can correspond to. An ordered pair o_1 can correspond to many possible edges in a given trace since a view is generated by a lossy transform. Assume that ordered pairs o_1 and o_2 correspond to sets of edges represented by s_1 and s_2 respectively. For example, given the ordered pair (Component:A, Component:B) and the trace from Figure 6.6, it would be mapped to a set containing the single edge - {(Time:10:11:05, Component:A), (Time:10:11:21, Component:B)}.

Next, we check if a cause-and-effect relationship exists between an edge in s_1 and one in s_2. If such a relationship is established, we discard the ordered pair corresponding to the effect while retaining its potential cause. We have reduced both the result set and the number of pairs to consider. We repeat this process for every pair of edges (that correspond to ordered pairs in the result set) in every trace until either we arrive at a single result or have explored all sampled traces.

Computing reachability is an expensive operation responsible for almost all of the time taken by ACT and is therefore applied after thresholding to reduce the

number of ordered pairs to be considered. The time taken to establish reachability is O($|r|^2 * n$), where $|r|$ is the number of ordered pairs in the result set and n is the number of traces sampled. In the worst case, it will be necessary to consider edges in all sampled traces if none of the calls in the result set are related to others. In practice, many calls are related and time to establish reachability is much lower than the worst case bound.

6.2.3 Application of ACT: An example

We now walk through an example of a simulated incident from the eBay dataset which demonstrates how techniques in ACT apply end to end and and highlights trade-offs SREs often need to make when an incident produces changes in a small number of traces. We simulate interruption in communication between PaymentService and OrderMgmtService. For users purchasing items, this call is required to validate the purchase. Interruption results in users being unable to place orders. Therefore, we want to highlight the missing call from PaymentService to OrderMgmtService. Assume that the probabilistic guarantee is 0.99 (if a call appears in more traces than the threshold, it is in the sampled traces with 99% probability) and t_{before} and $t_{incident}$ each contain 2K traces.

ACT computes a set of results, the elements of which are ordered pairs. The symmetric difference produces a result set of size 21 but after applying thresholding, the result set is empty. This implies that the sampled traces do not contain evidence of the correct execution, the changes produced by the incident, or both. SREs can now take two actions:

Reduce the threshold: An SRE may decide to trade-off number of results for time i.e. it is acceptable if the computed result has some irrelevant elements. The SRE will now choose a lower probability and re-compute the threshold. In

our example, the SRE chooses to drop the probability to 0.75. The result from symmetric difference contains 21 ordered pairs. The size of the result is now 11 after applying the new threshold. On applying reachability, we obtain a result of size 2 - the expected result and an additional, irrelevant suggestion.

Sample more traces: An SRE can also decide to trade-off time for number of results i.e. additional time is acceptable for fewer (ideally zero) irrelevant results. Since this is a simulated incident, we know that we can obtain the expected result with high probability by sampling 4K traces in each set. With the resampled traces, we compute the symmetric difference (result size is 41) and apply thresholding (result size reduced to 3). Applying reachability now yields exactly the expected result. The choice to trade-off time or number of results is situational - for example, if trading off number of results for time produces many false positives, SREs may pivot and sample more traces instead.

6.3 Evaluation

In Section 6.3.1, we discuss how the initial sample size is determined and used. To evaluate ACT, we simulate incidents based on how we expect traces to change for each incident category. Section 6.3.2 makes the case for simulating incidents and we discuss *how* we mutate traces. Section 6.3.3 describes the baselines we compare against. Finally, we compare the results of ACT with baseline techniques employed in prior work. We answer the following questions:

1. How is the initial sample size determined? (Section 6.3.1)
2. How do the results produced by ACT compare with baseline techniques? How do the individual techniques in ACT impact the results produced? (Section 6.3.4)

3. How does the time to obtain a result compare with baseline techniques? (Section 6.3.4)

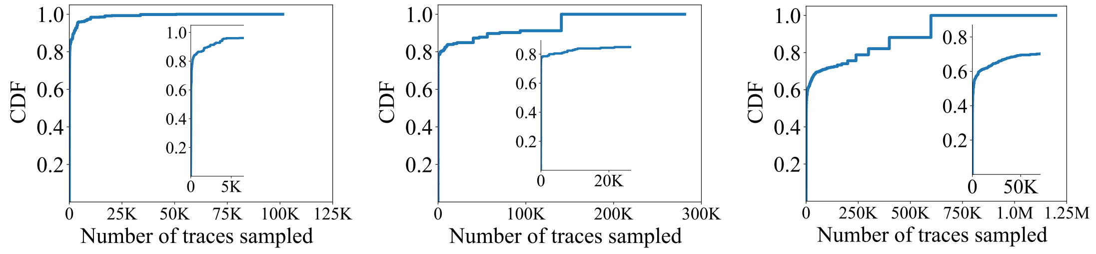
Finally, in Section 6.3.5, we present some highlights of integrating ACT with Jaeger [34] for online comparison of traces.

6.3.1 Determining the initial sample size

We discuss how we use ACT's probabilistic guarantees to determine the initial sample size from underlying traces and frequency statistics. This serves as an input when sampling t_{before} and $t_{incident}$ for localization.

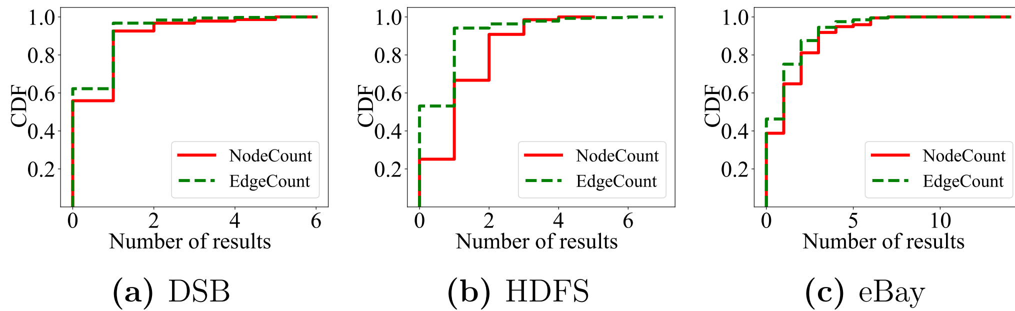
A structural change to a trace consists of ordered pairs that are missing during an incident which would normally be present in traces during steady state operation or additional calls that only occur during an incident. From our discussion of thresholding in Section 6.2.2, we provide a probabilistic guarantee that evidence of the correct interaction as well as structural changes to traces are represented in t_{before} and $t_{incident}$ respectively if they appear in more traces than the threshold.

We can plot a Cumulative Distribution Function (CDF) of the percentage of ordered pairs probabilistically guaranteed to be represented for a given number of sampled traces. Figure 6.8 depicts these for our three datasets. From the CDFs, we observe that although a very large number of traces would need to be sampled to identify every possible call (if it were missing), we find that a majority of calls can be identified with a sample that is orders of magnitude smaller. Accordingly, we sample 4K (of 20K) traces for DSB, 8K (of 60K) for HDFS and 20K (of 250K+) for eBay in our experiments.



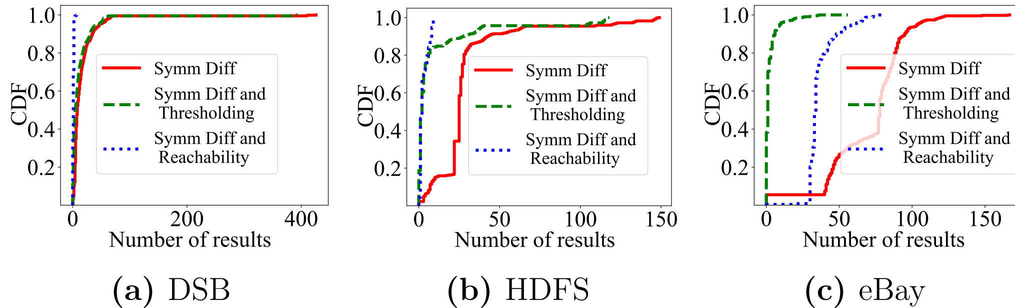
(a) DSB (22K traces) **(b)** HDFS (60K traces) **(c)** eBay(250000+ traces)

Figure 6.8: CDF of the number of traces to be sampled to identify any possible missing edge. The inlaid snippet of the CDF shows that a majority of calls can be identified with a sample that is orders of magnitude smaller.



(a) DSB **(b)** HDFS **(c)** eBay

Figure 6.9: For the cases when NodeCount and EdgeCount produce results, we plotted a CDF of number of results. ACT, meanwhile, produces exactly the expected answer for all of these cases.



(a) DSB **(b)** HDFS **(c)** eBay

Figure 6.10: CDFs of the number of results returned when we apply one or two techniques. Since the eBay dataset is noisy, symmetric difference and thresholding performs best, while symmetric difference and reachability generate the best results for DSB and HDFS. When all three techniques of ACT are applied, the result obtained is exactly the mitigation site.

Table 6.1: This table explains how we simulate the three failure modes we consider. For each, we describe the input, how traces are mutated and the expected output. We also specify the conditions that need to be satisfied in each case for a trace to be mutated. All mutated traces represent unsuccessful executions.

Incident Category	**Input**	**Condition for mutation**	**How are traces mutated?**	**Expected Result**
Component down	Randomly chosen component	Vertex corresponding to component is present in trace	Delete all edges to vertices corresponding to chosen component as well as the subgraph beneath each edge	Component chosen as input
Component Unreachable	Randomly chosen ordered pair	At least one edge corresponding to ordered pair is present in trace	Delete all edges corresponding to the chosen ordered pair as well as the subgraph beneath each edge	Ordered pair chosen as input
Buggy failure recovery	Randomly chosen ordered pair	At least one edge corresponding to ordered pair is present in trace	Delete all edges corresponding to the chosen ordered pair as well as the subgraph beneath each edge, then add an edge at each call site representing an attempt to recover from failure	Ordered pair chosen as input and additional ordered pair attempting recovery

6.3.2 Experimental Methodology

To conduct a quantitative evaluation of ACT using data from real incidents, we would have needed to collect traces during steady-state operation and then again when incidents occur. Although a large number of incidents occur (anecdotally, three or four every day), we are only interested in those in one of the categories described. Identifying these and capturing traces while they are still retained remains a challenge.

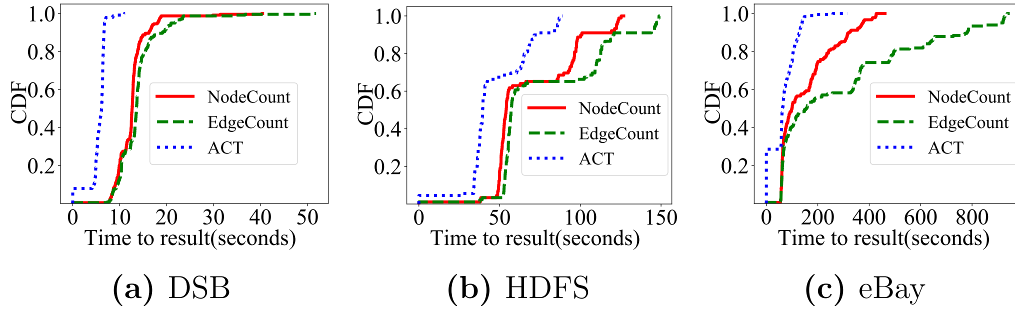
From our incident study and observations of traces generated when we inject

Table 6.2: ACT computes exactly the expected result for all but a few cases. In contrast, NodeCount and EdgeCount produce wrong answers for 30-50% of simulations. Answer = Set of localizations returned, Exact Answer = Answer is minimal, Superfluous Answer = Answer subsumes expected result, Wrong Answer = Answer does not contain expected result, No Answer = Answer is the null set.

		Number of simulations	**Exact Answer (%)**	**Superfluous Answer (%)**	**Wrong Answer (%)**	**No Answer (%)**
ACT	DSB	602	99.83 (601)	0.17 (1)	0	0
	HDFS	401	98.50 (395)	0.25 (1)	1.25 (5)	0
	eBay	418	99.76 (417)	0.24 (1)	0	0
Node Count	DSB	602	52.82 (318)	7.8 (47)	37.21 (224)	2.16 (13)
	HDFS	401	21.95 (88)	29.68 (119)	47.63 (191)	0.75 (3)
	eBay	418	25.11 (105)	21.77 (91)	48.80 (204)	4.41 (18)
Edge Count	DSB	602	58.47 (352)	2.66 (16)	36.38 (219)	2.49 (15)
	HDFS	401	63.59 (255)	4.49 (18)	31.17 (125)	0.75 (3)
	eBay	418	31.81 (133)	16.27 (68)	44.50 (186)	7.42 (31)

faults, we have a good grasp on how we expect the structure of traces to change for each incident category. Therefore, simulating incidents can serve as a good proxy. Simulation not only allows us to apply ACT to a wide range of scenarios but is also useful in testing its limits.

To simulate an incident, we randomly sample two sets of traces which we designate as t_{before} and $t_{incident}$ respectively. For each incident category, Table 6.1 describes inputs, *how* traces are mutated and expected output. Some fraction of traces in $t_{incident}$ that satisfy the condition for mutation are mutated to represent traces that would have been generated during the incident being simulated, while



(a) DSB **(b)** HDFS **(c)** eBay

Figure 6.11: CDFs of time taken to obtain a result. Reachability accounts for most of the time taken by ACT. Nodecount and Edgecount have highly variable time to result since trace of every unsuccessful execution needs to be compared with that of every successful execution and the number of unsuccessful executions can vary widely.

traces in t_{before} remain unmodified. All mutated traces represent unsuccessful executions. An unsuccessful execution is one for which we evaluate some external criteria and determine that the user request corresponding to the execution did not succeed. We choose simulations uniformly at random. t_{before} and $t_{incident}$ serve as inputs to the different techniques.

We use three trace datasets in our evaluation. These consist of a production dataset from eBay and two open-source datasets - DeathStar Benchmark (DSB) [73], a micro-services benchmark and Hadoop Distributed File System (HDFS) [72] traces. eBay has about 4500-5000 services, the dataset captures user requests as they purchase items during a week in November 2019. User requests to start a session and complete a purchase account for nearly two thirds of the requests; the remaining third is distributed across twenty other request types that span different system functions. Examples include changing user address and payment modes as well as updating items or item quantities. The captured requests record 250+ unique services and databases and 850+ unique calls. Vertices and edges represent services and calls between services respectively. DSB traces were generated by deploying the benchmark on a single machine and capturing traces

of different API types. HDFS traces were generated by deploying HDFS on a 9-node cluster and consists of traces obtained by reading and writing files of various sizes. The DSB and HDFS traces are in X-Trace [29] format and are captured at a lower level of abstraction where vertices represent execution of lines of code and edges represent the execution flow.

6.3.3 Baseline techniques

In prior work, graph analysis approaches [7, 89] transform graphs into vectors (by counting nodes or edges or converting them into strings) and compare pairs of graphs. The result returned is a pair of $\langle Successful, Unsuccessful \rangle$ traces such that they exercise the same execution path and are separated by the shortest distance. "Shortest" is precisely defined based on the distance metric and the representation used. NodeCount and EdgeCount represent traces as vectors containing the counts of components and calls between components respectively and use L_2 distance as the distance metric. Spectroscope [7] linearizes traces to produce an event string and uses string edit distance as its distance metric.

Since graph selection is not viable, we have adapted the different techniques to return the best result after comparing *all* pairs of traces. The inputs are vectors or string representations of traces in t_{before} and $t_{incident}$. For the resultant pair of traces, we compute the symmetric difference of the view of traces and apply reachability. This final step focusses attention on only the relevant results and is not employed in prior work. We take this step to be able to compare the results from the baselines with ACT. A single experiment comparing linearized traces took multiple *hours* as compared to the few seconds taken by other techniques. Hence, we ran simulations comparing ACT with NodeCount and EdgeCount only.

6.3.4 Results

Result Quality Table 6.2 summarizes the results for the simulations for which the change produced by the simulation is reflected in the sampled traces. We conducted hundreds of simulations for each dataset with the number of simulations for each incident category being roughly equivalent. As can be observed, ACT computes exactly the expected result for all but a few cases. In contrast, NodeCount and EdgeCount compute irrelevant results for 30-50% of simulations for which the changes are in the sampled traces.

Additionally, when NodeCount and EdgeCount produce the expected result (in 2.5% to 30% of the scenarios) depending on the technique and dataset, results include false positives. From our experiments, EdgeCount produces false positives in fewer scenarios than NodeCount. Figure 6.9 shows the CDF of the number of results produced by NodeCount and EdgeCount.

Impact of individual techniques To measure the impact of thresholding and reachability, we consider simulations for which ACT returns exactly the expected result, since the effects can be most clearly seen for these simulations. For the selected simulations, we employ combinations of one or two techniques and re-run them. Figure 6.10 visualizes the results we obtain. It is immediately apparent that computing symmetric difference with thresholding produces the best results for the eBay dataset indicating a noisier dataset than HDFS or DSB. Reachability plays a bigger role for DSB and HDFS datasets since these have more depth as compared with eBay dataset, in which graphs are wide and shallow. Across the board, the three techniques taken together are more powerful than any single pair of techniques.

Time taken to obtain result Figure 6.11 represents CDFs of time taken when the most number of traces are sampled for each dataset. Reachability computa-

tions account for almost all of the time taken by ACT. Symmetric difference and thresholding reduce the number of pairs for which reachability computations need to be performed - the time for which is negligible in comparison to reachability computation. ACT has a time bound of O($|r|^2 * n$), which is linear in the number of sampled traces, as discussed previously. The baseline techniques, however, have a time bound of O(s*u), where s is the number of successful executions in t_{before} and u is the number of unsuccessful executions in $t_{incident}$. Since the trace of every unsuccessful execution is compared with the trace of every successful execution, the time taken is quadratic.

6.3.5 Integrating with Jaeger: Implementation Details

We have integrated our approach with Jaeger [34] to enable online comparison of sets of traces. Jaeger is an open source, end-to-end distributed tracing tool that enables monitoring and troubleshooting complex distributed systems. It currently provides a feature that allows users to select and compare a pair of traces. The obvious drawback is that users need to know *which* traces to compare. We have extended the UI to compare sets of traces instead. Rather than requiring users to select traces as input, we accept as input the time since the incident started. This enables us to split the traces into before and after sets. For the purposes of our integration, we use HTTP status codes in the traces to mark them as successful or unsuccessful - a trace with any span returning a non-zero status code is considered unsuccessful. In general, SREs can use any criteria to label traces as successful or unsuccessful. With the two sets of traces and their labels as inputs, we extended Jaeger-UI to implement and visualize ACT.

Summary: ACT combines the use of aggregate and causal information in traces to effectively localize incidents. ACT identifies *exactly* the mitigation site in all

but a few cases. While witnessing a large number of traces is necessary to derive aggregate insights from traces, for a majority of incidents that produce structural changes in traces, the number of traces to be sampled is orders of magnitude smaller than the underlying traces captured by the system, making it viable for use in reducing the time to localize and thereby, resolve incidents.

6.4 Iterative Localization

Iterative localization naturally mimics how engineers approach problem diagnosis, starting with broad strokes and drilling down to specifics, making it a very natural extension of the techniques we have developed for localization. Since traces are a rich data source that can capture a wide range of contextual information, we would intuitively expect adding more information to produce better results. Iterative localization can improve localization in two ways by a) Providing more context that helps to eliminate some possible actions, thereby streamlining localization and b) Providing more specificity on where action is to be taken.

A natural question to ask is: Why not localize the incident by including all the information from the traces in one step? It turns out that including all the information in one step can distract from effective localization by producing irrelevant results, since each field added increases the cardinality of the space. Furthermore, since trace data can include a lot of context, they can be quite dense. Filtering traces by the results of prior localization reduces the data that would need to be processed at each step.

Iterative localization consists of multiple steps, adding more information at each step to improve the result. Figure 6.12 illustrates the process, which consists of projection, filtering, and localization. It also presents two examples of outputs produced after projection to different sets of fields and filtering by prior results.

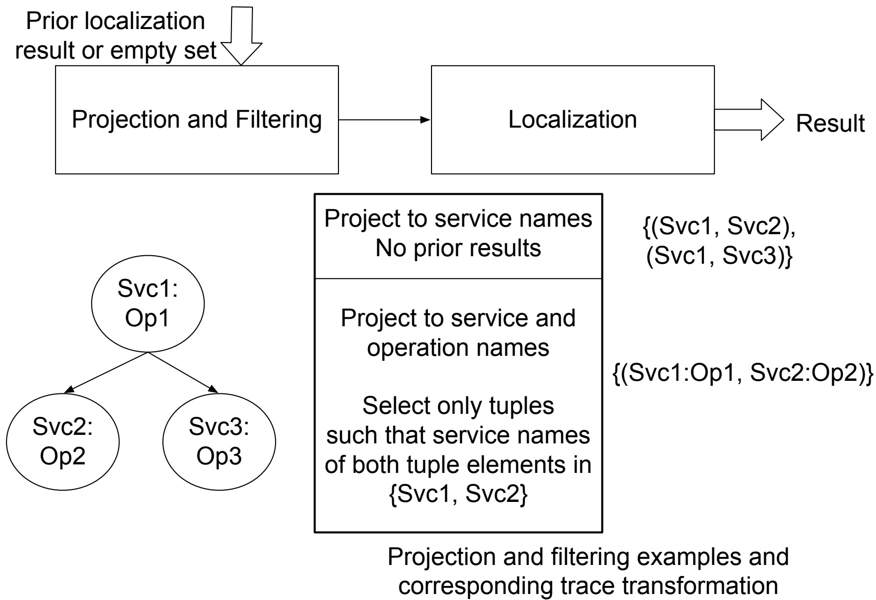


Figure 6.12: Iterative localization cycles through projection, filtering, and localization. We use ACT for localization, but projection and filtering can produce different results depending on the choice of fields to project down to and the results of prior localization. Two such examples are shown here.

Such projection, filtering, and localization can be repeatedly applied to obtain a solution containing targeted fields effectively. In our work, we first localize using only service names and subsequently add operation names and status of calls for more specific localization for the applications we consider. For a different application, other fields may be more appropriate. The fields chosen for iterative localization and the order in which they are explored require domain expertise and impact the results obtained.

We use sample open-source microservice applications [51] that encode failures previously described as having occurred in various production systems and gather traces both from steady state operation and after triggering the bug. First, we show how iterative localization can be used to streamline localization using a bug in the fallback path as an example. The bug is as follows: When the app-server attempts to write to db-primary and fails, it then attempts to write to

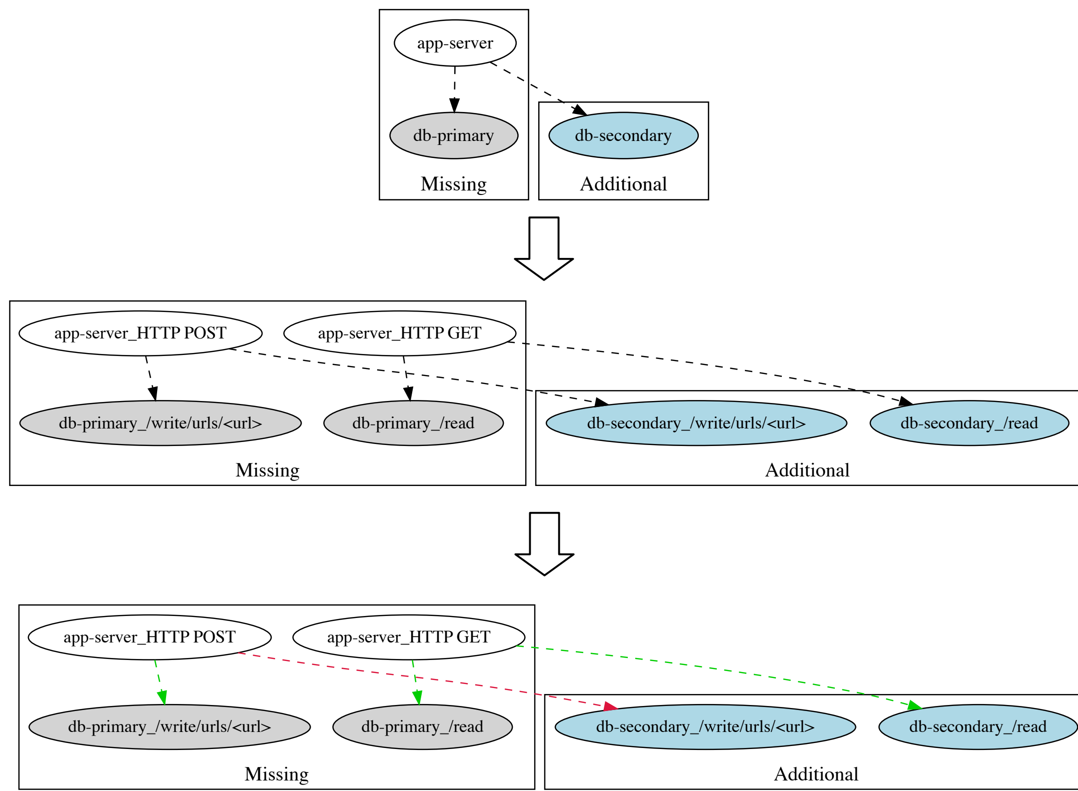


Figure 6.13: Shows the sequence of results produced by iterative localization for an example bug in the fallback path - when db-primary fails, db-secondary is invoked, but the call fails because of the lack of write permissions. The result of subsequent localizations are informed by and improve upon results of prior localizations. Legend: Dashed lines represent calls. Gray nodes represent a service or service:operationname that was in a successful execution but not in an unsuccessful execution; blue nodes represent the reverse. Finally, a green dashed line indicates a successful call while red indicates an unsuccessful call.

db-secondary, which also fails. Requests start failing en-masse and unexpected behavior is observed. We now present a series of results produced by iterative localization. As can be seen from the topmost result in Figure 6.13, we first determine that db-secondary is being called in the unsuccessful executions where db-primary was invoked in successful executions, so the next step is to compare calls to db-primary and db-secondary. In subsequent localizations, we observe that both read and write calls are made to db-primary and db-secondary in successful

and unsuccessful executions respectively, but based on the success or failure of individual calls (green represents success and red represents failure), engineers would also be able to deduce that the write call for db-secondary is the proximal cause of failures. Thus, in this instance, the addition of operation names and status of calls to service names helped streamline localization.

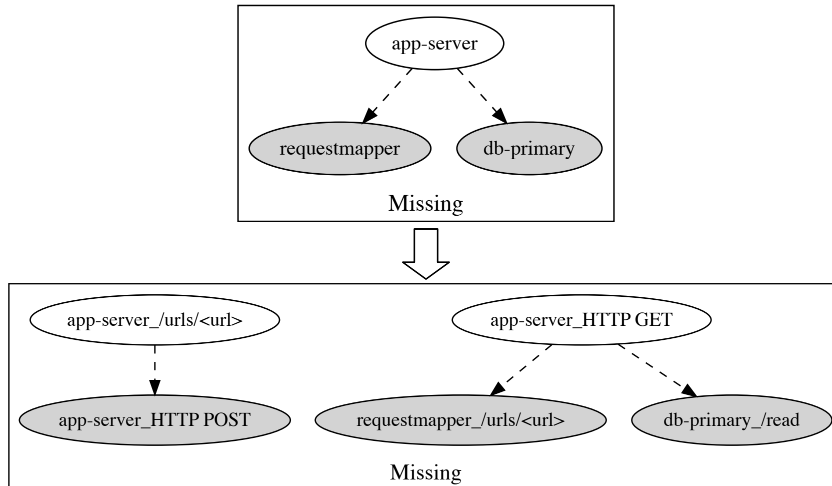


Figure 6.14: Shows that even when iterative localization does not streamline results, it can add specifics that help engineers take action. In this case, the bug is that the call to requestmapper was critical but not recognized as such and an RPC failure from app-server to requestmapper caused failure of the request as a whole. Legend: Dashed lines represent calls. Gray nodes represent a service or service:operationname that was in a successful execution but not in an unsuccessful execution

Iterative localization can also be used to obtain more specifics on where action is to be applied. Here, we will use a bug triggered by an RPC failure as an example. The bug is as follows: The RPC call from app-server to requestmapper failed and since this call was critical, but not recognized as such, its failure led to overall failure of the request and this pattern was repeated across requests. Since the call from app-server to requestmapper failed, the subsequent call from app-server to db-primary was never made. As we can see from Figure 6.14, based on the traces, it appears that the failure arises due to one or both missing calls

from the app-server service. By adding operation names, we are able to see the specific operations on the two services that are missing. Services can be associated with many operations. Therefore, identifying missing calls to specific endpoints is very useful. In this case, engineers will need to check two calls: (app-server:GET, requestmapper/urls/<url>) and (app-server:GET,db-primary/read).

We have illustrated by example that iterative localization is useful for streamlining results of prior localization as well as adding more specificity to the results obtained. Our preliminary results are promising and there are several directions for future work. One is to extend the current work to a large scale setting and evaluate the quality of results obtained. Another direction is to evaluate the various trade-offs in iterative localization - additional time for iteration and improved efficiency by reducing data processed versus additional traces required to achieve the same results, for example.

We have shown how having different observability signals from systems can influence which problems are solved. With Nemo, we explored the limits of our approach in a perfect information scenario for debugging systems. While most large systems today do not gather perfect information, many produce distributed traces, making ACT highly viable. Since collecting and maintaining perfect information at scale is challenging and cost prohibitive, building system observability (traces that are more fine-grained with richer context, for example) that can solve a variety of problems by varying their level of detail can be extremely powerful. In iterative localization, we relaxed the constraint that projects down information in traces to only service or component names. We continue this theme in the next chapter in which we describe nascent work in identifying instances of common design patterns based on empirical observations.

Chapter 7

Identifying Distributed Systems Behaviors

It is common practice to build microservices using well-understood design patterns [90–94]. Examples include patterns for caching, fallbacks, and retries. A fallback, if configured, is invoked after an initial call to a service fails. Similarly, when the requested data is not found in the cache, a cache miss occurs and an additional call to the database is made. While patterns for fallbacks or caches are well-understood, their instantiation for a particular system is typically unknown.

Mining design patterns is an active area of research and has applications in program comprehension, feature identification, feature extraction, and assessing software quality. Instantiations of application-level patterns can also be used for debugging behavioral and performance issues. Recent work explores mining the architecture of microservice applications based on their Kubernetes deployments [95] to test if applications adhere to microservice design principles and to refactor them as necessary.

While most prior work has focused on finding instances of such patterns by analyzing source code via static and dynamic analysis [96–100], there is a missing

piece - finding patterns in source code does not guarantee that the system behaves as programmers intend them to. Additionally, engineers may not recognize instances of design patterns due to the scale and constantly evolving nature of systems, exacerbating the problem.

To gain confidence that systems behave as expected, we look for instances of design patterns in observations of executions. In our work, we consider patterns that arise from communication between participants in distributed executions. To find such instances of design patterns, we need to reason in aggregate across many executions. For example, when two services occur in mutually exclusive executions, we can only identify this pattern by observing pairs of executions. Additionally, observing many executions allows us to trim false positives. If we postulate that service X and Y occur in mutually exclusive executions and subsequently observe a single execution containing calls to both service X and service Y, the instance is disqualified and not returned as a result that matches the pattern.

Our key insight is that, after factoring out application-specific details, querying observations of executions allows us to match templates of design patterns to their instantiations across different applications. Application experts select and transform fields from traces into sets of tuples that are loaded into database tables based on a predefined schema. Then, pattern experts write SQL queries that are run against a database containing trace data from many executions. We describe our methodology in detail in the next section. We find instances of common design patterns - caching and fallbacks in sample microservice applications (HipsterShop [101], Deathstarbench [73], and applications from the Filibuster [51] corpus). Our preliminary results are promising and warrant further work in this space. In Section 7.1, we present our methodology with the fallback pattern as an example and discuss the system requirements that need to be satisfied for

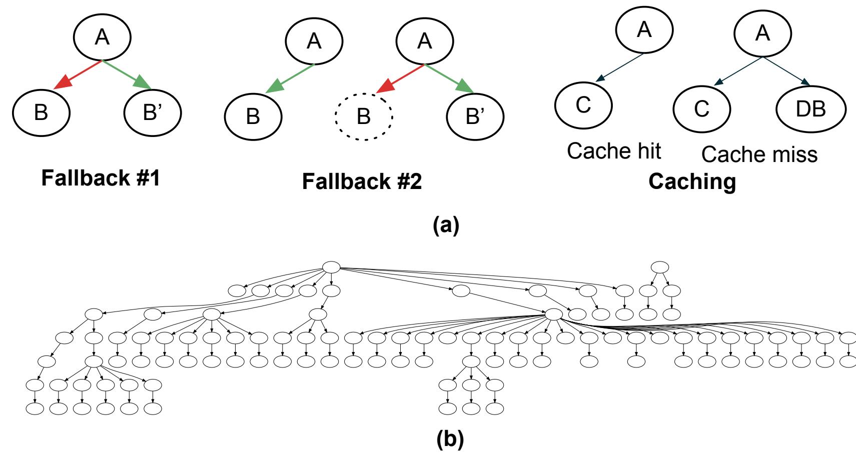


Figure 7.1: (a) represents common design patterns such as fallbacks and caching effects, where the red and green arrows represent failed and successful calls respectively. The dotted lines represent a service to which a call was attempted, but the message was dropped or lost in transit. (b) is an example trace taken from a real production system

our techniques to apply. In Section 7.2, we discuss our results and Section 7.3 highlights the applicability of our methodology to different problem domains.

7.1 Methodology

In our work, we focus on detecting two patterns in distributed execution traces: fallbacks and caching effects, templates of which are represented in Figure 7.1 (a). A fallback may be invoked in two main contexts. When a call returns an error in response to which the caller then makes a call to a different service, which can be observed by looking at a single execution. In the template, potential fallbacks are identified by a failed call from service A to service B, followed by a successful call from service A to some other service, B', at a later time.

Alternatively, a fallback may be invoked when a call is dropped or lost in transit and triggers a timeout on the caller, which then makes a call to a different service. For this type of behavior, we need to look at *at least* two execution traces

to confirm that this effect is indeed a fallback. In this template, the first successful execution contains a successful call, made from service A to service B. Contrast this with another successful execution, in which service A attempts to call service B but fails. Later, service A successfully calls service B'. Either call can occur, but not both, and it may be the case that neither of them occur.

We are also able to differentiate between cache hits and misses when observing executions. A cache hit occurs when a service attempts to retrieve data from cache and the data is present. A cache miss occurs when the data is not present in the cache, and the service must then retrieve data from the database at a later time. The basic template for cache miss is similar to fallback in single execution, but no call failures occur - instead, data is not found so more operations are needed to retrieve data.

While these patterns are simple, looking for them in practice is difficult, as distributed traces are often very large. Figure 7.1 (b) is a real (anonymized) trace from a production system. Since the templates of patterns we would like to find are small compared with individual traces, matching templates to instances of their occurrence manually is impractical. Next, we describe our methodology to automatically find instances of patterns.

Our methodology to discover instances of patterns has two phases: an application specific phase to normalize trace data, and a query phase, which returns pattern instances. In the first phase, an application expert identifies the fields in the trace which need to be selected, transformed or discarded. Application experts always select fields such as service names, operation names, and error codes, but these may be named differently in various applications. Fields corresponding to service instance names, method names, file or line numbers may also be selected depending on the data recorded in the traces and the pattern whose

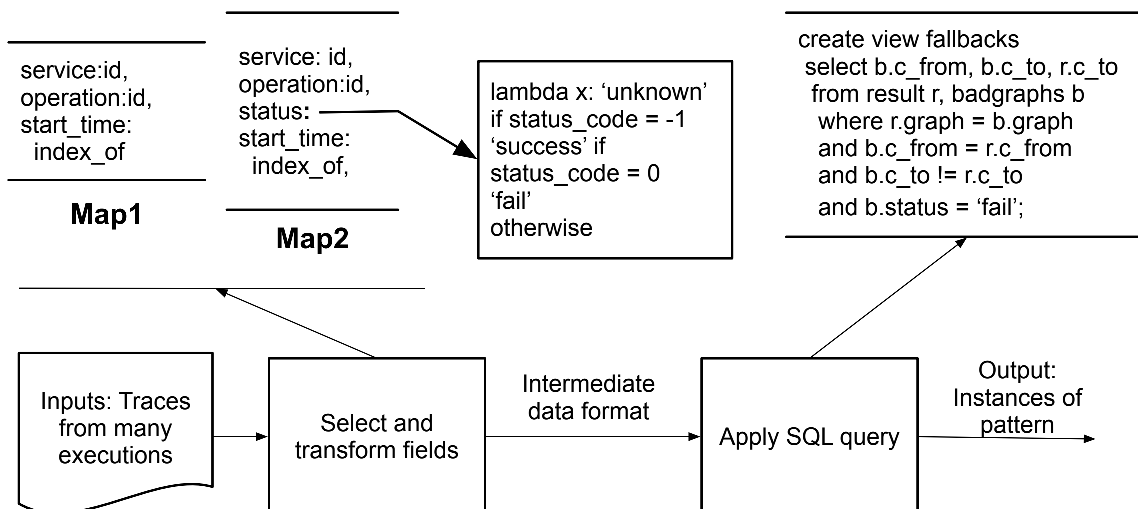


Figure 7.2: System workflow showing the steps in our methodology with a running example. The id in the mappings corresponds to identity, which means that the fields are retained as-is. index_of indicates that the start time is converted into a logical time and we have also shown how status code is mapped to one of three strings.

instantiations we are attempting to discover. A database containing normalized data corresponding to the set of executions is produced as output.

For example, all applications analyzed in this work utilize Jaeger tracing [34]. We have found that mining patterns in Jaeger traces requires selecting the service name, operation name, and status code labels, and transforming timestamp labels to logical time. Transformation of timestamp labels is encoded in the “index_of" function shown in Figure 7.2 (Map2). Applying this mapping to all traces studied prepares them for further analysis in the subsequent phases. Application-specific expertise is required to write mappings for other applications to uncover these same patterns. We process the transformed trace data and load information about each event and call in each trace in the corpus into SQL tables. The result is a normalized data format which allows us to easily execute queries in the next phase.

In the second, query phase, a pattern expert writes queries in SQL to identify

the patterns we are interested in from sets of traces such that the same query can be applied to processed trace data from different applications. For example, the fallback in a single execution can be identified by a failed call from service A to B followed by a later successful call from A to B'. A SQL query can identify this by selecting pairs of events with the following characteristics: exactly one failure and one success as sibling nodes, in which the failure occurs temporally before the success. A snippet of this query is shown on the extreme right in Figure 7.2.

To find instantiations of design patterns in traces of executions, we require that any pair of system executions is differentiated by at most one change, for a given set of executions. In our setting, a single change translates to failure of a call or crash of a service instance. This requirement is necessary since different changes can interact with each other leading to false positives that we cannot disambiguate. To satisfy this requirement, we are exploring a framework that runs end-to-end tests repeatedly in a staging environment killing a process, injecting delays or mocking failures in different runs. Our framework also corrects for false positives as a result of non-deterministic effects of executions by witnessing traces of many executions and ordering results by decreasing frequency of their occurrence.

For our analysis, we consider sample microservice applications integrated with distributed traces - HipsterShop, Deathstarbench and applications in the Filibuster corpus. In the next section, we discuss our experimental methodology as well as preliminary results that identify potential fallbacks and caching effects in different applications.

7.2 Evaluation

To find instances of fallbacks and caching templates in different applications, the set up consists of a few steps. First, we configure and run applications so that

traces of executions are captured. Secondly, we identify functional tests to run or APIs to invoke that exercise desired functionality. We run the functional test or invoke the API at least once to capture traces during normal operation. Finally, we trigger fallback or caching behavior in applications via injecting crash faults or mocking errors in responses and run the functional test again, analyzing traces captured from executions during normal system operation and when different faults are injected.

Table 7.1: Instances of patterns in different applications

	Fallback 1	**Fallback 2**	**Caching**
Hipstershop	✓	✓	
Cinema-6 (Filibuster)	✓	✓	
Netflix (Filibuster)	✓	✓	
Expedia (Filibuster)	✓	✓	
Deathstarbench			✓

In our setting, we have configured applications to send traces to Jaeger. As discussed in the previous section, we select the service name, operation name, and status code labels from each trace, and transform timestamp labels to logical time. Keeping these mappings fixed, we write different queries for each template we want to identify. As can be seen in Table 7.1, we found evidence of the two fallback patterns in several applications in the Filibuster corpus and were also able to confirm that the fallbacks we added programmatically to Hipstershop were discovered by our queries in executions. We also found evidence of caching effects in Deathstarbench executions captured by crashing instances of different caches and invoking specific APIs that reveal cache hits and misses.

7.3 Discussion

Our work in mining specific, common design patterns from distributed traces is unique, efficient, and covers a space of distributed design patterns research that is less explored in previous work. The system behaviors identified in this work can have a variety of applications, most notably in building domain knowledge, feature development, and debugging behavioral and performance issues. Some examples are:

1. If we determine that some service X can serve as a fallback for both Y and Z, but a fallback has not yet been configured for Z, developers may configure X as a fallback for Z as well. Alternatively, if Z fails, traffic may be temporarily redirected to X to keep the system functional.

2. If an increase in cache misses is observed at the same time as a performance regression, investigating the cache would be a good place to start.

3. Finding examples of anti-patterns can help engineers proactively identify and fix issues before they cause a failure.

Retries follow a similar template to fallbacks within a single execution, except both calls are to the same destination, with the earlier call having failed. However, when querying for this pattern, we found that retries are observed when errors propagate up the trace graph when a fallback is invoked not by the immediate caller, but a service higher up. We speculate that retries that occur independently of fallbacks could represent an anti-pattern, especially if the retry is to the same service instance as the failed call.

Our first phase requires that application experts select and transform fields in traces to obtain a common set of labels that can be queried; a manual and

tedious process. We posit that we can address this issue by automating the mapping process shown in Figure 7.2. These mappings are simple examples of trace abstraction, in which the size and complexity of traces are reduced by eliminating low-level details and preserving causal relationships and necessary information for trace comprehension. Successful usage of trace abstraction would allow our tool to tolerate variations in traces across applications and discrepancies within traces due to non-deterministic effects. A trace abstraction-like approach has been used to mine patterns from traces to account for dynamic program behavior [96]. We seek to develop a unified approach to trace abstraction in future work.

In this chapter, we have described our techniques to identify instances of design patterns. We have employed these successfully to identify potential fallbacks and caching effects in several different applications. As discussed previously, finding these provides some evidence that the system functions as expected and has a variety of applications in building domain knowledge, feature development, and debugging. Future work in this space involves identifying and writing queries for more such patterns and anti-patterns as well as evaluating our techniques in a larger setting. Automatically finding mappings in the application-specific phase to reduce manual work for application experts is a challenging direction of future work as well. In the next chapter, we conclude by summarizing our findings in the previous chapters and describing several directions for future work.

Chapter 8

Conclusion

In the previous chapters, we have shown how we can understand, improve, and troubleshoot systems by asking and answering questions of observations of system executions. Further, we have shown how the choice of data format - data provenance and distributed traces - dictates the problems that can be solved. Data provenance represents a perfect information scenario and having access to provenance graphs allows us to explore the limits of problems that can be solved. Distributed tracing has seen increased adoption in industry and solutions that use traces need to address challenges of using observations from real systems.

There are several avenues for future work. Relaxing our assumptions can bring to the fore challenging problems:

- Input generation: A variety of approaches to input generation and test generation (KLEE [102], Quickcheck [103]) are available. Since some fault tolerance bugs in distributed systems are triggered only by specific interleavings of inputs and fault events; eg. Zave's counterexamples to the correctness invariants for Chord [104], work that co-optimizes the search through faults and inputs is an interesting direction for future work.

- Non deterministic scheduling orders: Verification techniques such as model checking - particularly the software model checkers capable of verifying real implementations - are ideally suited for reasoning about bugs triggered by non-deterministic scheduling orders. Recent work on semantic-aware software model checkers (e.g. SAMC [105]) is particularly encouraging. However, these tools require encoding domain knowledge about any independence and symmetry characteristics 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. Combining an approach like LDFI [15] that builds models of domain knowledge with tools such as SAMC would allow us to reason about failures at the intersection of partial failure and asynchrony.

- Gray failures: In Chapter 6, we troubleshoot bugs that arise from crashes and message drops. However, gray failures are increasingly common [18,106]. Consider, for example, a call between instances of A and B fails for some inputs. In this scenario, it will be the case that the call from A to B is missing in some percentage of traces. Developing techniques to diagnose or localize gray failures is an important direction of future work.

In the long run, we would like to unify the techniques used for incident localization and debugging. This requires rich observability - fine grained traces rich in context would be an example. We can then envision applying the same techniques to different views of the observations and solve different problems. We see two obvious sub-problems that need to be addressed to do so. First, in most of our work, we have projected down to service name and file:line numbers. In the later chapters, we have relaxed this to include operation names and status of calls. However, traces can contain an arbitrary number of fields and one direction

of future work would be to determine which projections retain useful information about the causality of event interactions that can be used in solving distributed systems problems. Second, in Chapter 6, we presented some initial results with respect to the usefulness of iterative localization. Evaluating this for large scale systems and combining it with identifying useful projections would be the first step towards an unified approach to troubleshooting distributed systems.

In Chapter 7, we have highlighted the wide applicability of identifying distributed systems behaviors and described how we do so via SQL queries. We observe, however, that system topologies are broadly fixed and industry standard best practices follow set templates. Furthermore, services and their fallbacks serve "essentially" the same function in the context they are invoked. While a call to a cache or the underlying database may differ in performance, they both are expected to return values that are generally consistent with each other. Recent work [107, 108] has explored the use of word embeddings in the context of graphs. Since DAGs are the most general representation of traces, natural language processing techniques represent a powerful alternative to identifying template instantiations that is worth exploring.

In the introduction, we presented anomaly detection and debugging performance failures as examples of problems that could be addressed via querying system observations. Although some prior work exists in this space - Sifter [109] performs anomaly detection with distributed traces as inputs to achieve representative sampling of traces and Zeno [44] has introduced the concept of temporal provenance to diagnose performance problems - these problems are far from solved and would benefit from work in this space.

With the ubiquity of distributed systems, the adoption of the public cloud and the heterogeneity of software solutions, peeking into or modifying system

components may be challenging. As a result, asking and answering questions over unmodified system observations is uniquely suited to solve distributed systems problems in this setting. In this thesis, we have demonstrated the suitability of this approach by developing techniques that successfully address three such problems subject to our constraints - exploring the fault tolerance space, troubleshooting systems and identifying behavior patterns in distributed systems executions. We are hopeful that our work will pave the way for research into other problems that may be solved via querying system observations.

Bibliography

[1] Songyun Duan, Shivnath Babu, and Kamesh Munagala. Fa: A system for automating failure diagnosis. In *2009 IEEE 25th International Conference on Data Engineering*, 2009.

[2] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, ICST '18, 2018.

[3] IBM. Fault localization in cloud systems using golden signals. 2021. [Online; accessed August 2021].

[4] Hanzhang Wang, Phuong Nguyen, Jun Li, Selcuk Kopru, Gene Zhang, Sanjeev Katariya, and Sami Ben-Romdhane. Grano: Interactive graph-based root cause analysis for cloud-native distributed data platform. *Proc. VLDB Endow.*, 12(12), 2019.

[5] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. Groot: An event-graph-based approach for root cause analysis in industrial settings. ASE '21, 2021.

[6] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems. *Commun. ACM*, 59(8), 2016.

[7] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11. USENIX Association, 2011.

[8] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI '04. USENIX Association, 2004.

[9] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[10] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. Lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[11] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Mochi: Visual log-analysis based tools for debugging hadoop. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud '09. USENIX Association, 2009.

[12] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV. Association for Computing Machinery, 2010.

[13] Kamala Ramasubramanian, Kathryn Dahlgren, Asha Karim, Sanjana Maiya, Sarah Borland, Boaz Leskes, and Peter Alvaro. Growing a protocol. In *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'17. USENIX Association, 2017.

[14] The netflix simian army `https://medium.com/netflix-techblog/fit-failure-injection-testing-35d8e2a9bb2`, 2014.

[15] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15. Association for Computing Machinery, 2015.

[16] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. Modeling the parallel execution of black-box services. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '11. USENIX Association, 2011.

[17] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of*

the Thirteenth EuroSys Conference, EuroSys '18. Association for Computing Machinery, 2018.

[18] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18. USENIX Association, 2018.

[19] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. Association for Computing Machinery, 2009.

[20] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18. Association for Computing Machinery, 2018.

[21] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17. Association for Computing Machinery, 2017.

[22] Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi, and Luis Ceze. Troubleshooting transiently-recurring problems in production systems with blame-proportional logging. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18. USENIX Association, 2018.

[23] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18. Association for Computing Machinery, 2018.

[24] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI. Association for Computing Machinery, 2011.

[25] Soila P. Kavulya, Scott Daniels, Kaustubh Joshi, Matti Hiltunen, Rajeev Gandhi, and Priya Narasimhan. Draco: Statistical diagnosis of chronic

problems in large distributed systems. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, 2012.

[26] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016.

[27] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15. Association for Computing Machinery, 2015.

[28] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[29] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI '07. USENIX Association, 2007.

[30] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI '06. USENIX Association, 2006.

[31] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. Association for Computing Machinery, 2017.

[32] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06. Association for Computing Machinery, 2006.

[33] Open Telemetry. https://opentelemetry.io//. [Online; accessed August 2021].

[34] Jaeger Tracing. https://www.jaegertracing.io/. [Online; accessed August 2021].

[35] Lightstep. https://www.lightstep.com/. [Online; accessed August 2021].

[36] NewRelic. https://newrelic.com/. [Online; accessed August 2021].

[37] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10. Association for Computing Machinery, 2010.

[38] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10. Association for Computing Machinery, 2010.

[39] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07. Association for Computing Machinery, 2007.

[40] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06. USENIX Association, 2006.

[41] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01. Springer-Verlag, 2001.

[42] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2), 2000.

[43] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Differential provenance: Better network diagnostics with reference events. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV. Association for Computing Machinery, 2015.

[44] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI '19. USENIX Association, 2019.

[45] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for software-defined networks. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI '17. USENIX Association, 2017.

[46] Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M. Hellerstein. Debugging distributed systems with why-across-time provenance. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18. Association for Computing Machinery, 2018.

[47] Lennart Oldenburg, Xiangfeng Zhu, Kamala Ramasubramanian, and Peter Alvaro. Fixed it for you: Protocol repair using lineage graphs. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, CIDR '19, 2019.

[48] Kamala Ramasubramanian, Ashutosh Raina, Jonathan Mace, and Peter Alvaro. Act now: Aggregate comparison of traces for incident localization, 2022.

[49] Netflix. Chaos Monkey. `https://netflix.github.io/chaosmonkey/`. [Online; accessed August 2021].

[50] Gremlin. `https://www.gremlin.com/`. [Online; accessed May 2022].

[51] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21. Association for Computing Machinery, 2021.

[52] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 3milebeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21. Association for Computing Machinery, 2021.

[53] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[54] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '14. USENIX Association, 2014.

[55] Wei Lin, Mao Yang, Lintao Zhang, and Lidong Zhou. PacificA: Replication in Log-Based Distributed Storage Systems. Technical report, 2008.

[56] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI '04. USENIX Association, 2004.

[57] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1993.

[58] Karthekeyan Chandrasekaran, Richard Karp, Erick Moreno-Centeno, and Santosh Vempala. Algorithms for implicit hitting set problems. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11. Society for Industrial and Applied Mathematics, 2011.

[59] Nemo. `https://github.com/numbleroot/nemo`. [Online; accessed August 2021].

[60] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16. Association for Computing Machinery, 2016.

[61] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16. Association for Computing Machinery, 2016.

[62] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo Seltzer. Frappuccino: Fault-detection through runtime analysis of provenance. In *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '17. USENIX Association, 2017.

[63] Ang Chen, Yang Wu, Andreas Haeberlen, Boon Thau Loo, and Wenchao Zhou. Data provenance at internet scale: Architecture, experiences, and the road ahead. CIDR '17, 2017.

[64] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14. Association for Computing Machinery, 2014.

[65] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76. IEEE Computer Society Press, 1976.

[66] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog '10. Springer-Verlag, 2010.

[67] JD Ullman. *Principles of Database and Knowledge-Base Systems, volume II.* WH Freeman, 1990.

[68] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06. Association for Computing Machinery, 2006.

[69] Molly. `https://github.com/palvaro/molly`. [Online; accessed August 2021].

[70] Amazon goes down, loses $66240 per minute. `https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/?sh=298ad330495c`, 2013. [Online; accessed August 2021].

[71] 3 minute outage costs Google $545000 in revenue. `https://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/`, 2013. [Online; accessed August 2021].

[72] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10. IEEE Computer Society, 2010.

[73] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19. Association for Computing Machinery, 2019.

[74] Marisa R. Grayson. Cognitive work of hypothesis exploration during anomaly response. *Queue*, 17(6), 2020.

[75] J. Paul Reed. Beyond the fix-it treadmill. *Queue*, 17(6), 2020.

[76] Richard I. Cook. Above the line, below the line. *Queue*, 17(6), 2020.

[77] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19. Association for Computing Machinery, 2019.

[78] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16. Association for Computing Machinery, 2016.

[79] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14. Association for Computing Machinery, 2014.

[80] Yaohui Wang, Guozheng Li, Zijian Wang, Yu Kang, Yangfan Zhou, Hongyu Zhang, Feng Gao, Jeffrey Sun, Li Yang, Pochian Lee, Zhangwei Xu, Pu Zhao, Bo Qiao, Liqun Li, Xu Zhang, and Qingwei Lin. Fast outage analysis of large-scale production clouds with service correlation mining. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21. IEEE Press, 2021.

[81] Junjie Chen, Xiaoting He, Qingwei Lin, Yong Xu, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. An empirical investigation of incident triage for online service systems. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19. IEEE Press, 2019.

[82] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. Continuous incident triage for large-scale online service systems. In *Proceedings of the 34th*

IEEE/ACM International Conference on Automated Software Engineering, ASE '19. IEEE Press, 2019.

[83] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI '18. USENIX Association, 2018.

[84] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive realtime datacenter fault detection and localization. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI '17. USENIX Association, 2017.

[85] OpenTracing. `https://opentracing.io/`. [Online; accessed August 2021].

[86] OpenCensus. `https://opencensus.io/`. [Online; accessed August 2021].

[87] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS '03. USENIX Association, 2003.

[88] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '20. Association for Computing Machinery, 2020.

[89] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02. IEEE Computer Society, 2002.

[90] Chris Richardson. *Microservice Patterns. With Examples in Java*. Manning, Shelter Island, NY, 2019.

[91] Cloud design patterns. `https://docs.microsoft.com/en-us/azure/architecture/patterns/`. [Online; access May 2022].

[92] Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. Designing microservice systems using patterns: An empirical study on quality trade-offs. 2022.

[93] Felipe Osses, Gastón Márquez, and Hernán Astudillo. Exploration of academic and industrial evidence about architectural tactics and patterns in microservices. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18. Association for Computing Machinery, 2018.

[94] Gastón Márquez, Mónica M. Villegas, and Hernán Astudillo. A pattern language for scalable microservices-based systems. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ECSA '18. Association for Computing Machinery, 2018.

[95] Jacopo Soldani, Giuseppe Muntoni, Davide Neri, and Antonio Brogi. The μtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience*, 51, 2021.

[96] Chun-Tung Li, Jiannong Cao, Chao Ma, Jiaxing Shen, and Ka Ho Wong. An agnostic and efficient approach to identifying features from execution traces. *Knowledge-Based Systems*, 2022.

[97] B. Bafandeh Mayvan, A. Rasoolzadegan, and Z. Ghavidel Yazdi. The state of the art on design patterns. *J. Syst. Softw.*, 125(C), 2017.

[98] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Inferring hierarchical motifs from execution traces. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18. Association for Computing Machinery, 2018.

[99] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03. IEEE Computer Society, 2003.

[100] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Detecting the behavior of design patterns through model checking and dynamic analysis. *ACM Trans. Softw. Eng. Methodol.*, 26(4), 2018.

[101] OpenCensus. Hipster Shop: Cloud-Native Microservices Demo Application. `https://github.com/census-ecosystem/opencensus-microservices-demo`.

[102] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08. USENIX Association, 2008.

[103] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00. Association for Computing Machinery, 2000.

[104] Pamela Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, 42(2), 2012.

[105] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[106] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18. USENIX Association, 2018.

[107] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. 2017.

[108] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. 2016.

[109] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. Association for Computing Machinery, 2019.