Fixed It For You: Protocol Repair Using Lineage Graphs

Lennart Oldenburg[1,2], Xiangfeng Zhu[1], Kamala Ramasubramanian[1], and Peter Alvaro[1]

[1]University of California, Santa Cruz

[2]Technische Universität Berlin

{loldenbu, xzhu27, kramasub, palvaro}@ucsc.edu

Abstract

Distributed systems are difficult to program and near impossible to debug. Existing tools that focus on single-node computation are poorly-suited to diagnose errors that involve the interaction of many machines over time. The database notion of provenance would appear to be a better fit for answering the sort of cause-and-effect questions that arise during debugging, but existing provenance-based approaches target only a narrow set of debugging scenarios. In this paper, we explore the limits of provenance-based debugging. We propose a simple query language to express common debugging questions as expressions over provenance graphs capturing traces of distributed executions. When programs and their correctness properties are written in the same high-level declarative language, we can go a step further than highlighting errors by often generating repairs for distributed programs. We validate our prototype debugger, Nemo, on six protocols from our taxonomy of 52 real-world distributed bugs, either generating repair rules or pointing the programmer to root causes.

1. INTRODUCTION

Distributed systems permeate our world but we are only just beginning to understand how to program and manage them. The challenges of programming and reasoning about orchestration of long-running computations that span large numbers of independent physical machines while tolerating partial failure and unpredictable delay [6, 13] are the exact same factors making them seriously difficult to debug. Conventional approaches such as process-centric debuggers are of little help, as the observed effects of distributed bugs are often remote from their causes in space and time.

Once again, the database community is in a position to advance the state of the art in distributed systems by dusting off an old database idea: *data provenance* [8]. We strongly believe the high-level, data-centric *explanations* of computations obtained via provenance collection to be the right way to debug errors spanning multiple communicating machines. Recently, we have seen tantalizing evidence (sometimes at venues such as CIDR!) that provenance could become the basis of the debugging tool sets of the future [11, 29, 10, 28].

Existing provenance-based debugging approaches such as differential provenance [11] (highlighting code and data regions where an execution went wrong) are ideal for identifying the causes of a class of programmer errors we call *errors of commission.* Errors such as specifying an incorrect state transition, a faulty configuration line, or an off-by-one loop bound are common in all programs, but in distributed systems are sometimes triggered only by rare events (e.g., crashes and message reordering). Because repairing such bugs is easy once these lines have been identified, techniques like differential provenance are extremely effective at debugging them.

Unfortunately, nearly as common in large-scale distributed systems are *errors of omission*, as we quantify in Section 4.1. Here, the problem is not so much a mistake but an oversight: the programmer has insufficiently developed the protocol. Examples include insufficient synchronization between communicating processes (leading to race conditions) and insufficient redundancy (e.g., retry and replication) to ensure availability and durability in the face of faults. State-of-the-art provenance-based debugging is of no help to the programmer here, because there is no offending line of code to point to!

In this paper, we explore the limits of provenance-based forensics by recasting debugging as a question-and-answer process over provenance graphs that represent traces of distributed executions (successful and failed). Debugging questions, e.g., “what do all successful runs have in common?” or “how did this failed run differ from this other failed run?”, are posed as *graph queries* using a simple algebraic language with familiar set-theoretic operators. We show how the question of differential provenance (“how does the failed run differ from successful ones?”) is subsumed by this approach. The answers to all of these questions identify code regions to present to a programmer for further study, and are only effective against errors of commission.

We then show the surprising result that when programs are appropriately constrained, we can go a step further and in many cases *repair* errors of omission. The key idea is that when programs and their specifications are written in the same provenance-enhanced relational logic language, we can co-analyze the provenance of system state with the provenance of the specification predicates. Because the specification describes the non-distributed behavior of the program,

9th Biennial Conference on Innovative Data Systems Research (CIDR ‘19)
January 13-16, 2019, Asilomar, California, USA.

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

Figure 1: Asynchronous primary/backup ("Async P/B") replication protocol in Dedalus. Persistent relations in bold.

it guides the generation of code changes that correct the distributed program towards compliance with its sequential specification. These program modifications often amount to a few lines of code, avoiding the complexities of combinatorial program synthesis.

We make the following contributions:

- We propose a query language for expressing debugging questions as queries over provenance graphs representing distributed program executions. We show how a number of debugging best practices can be expressed by this paradigm, including differential provenance.
- We provide a new taxonomy for 52 real-world distributed bugs from large-scale distributed systems [18], determining for each whether our framework can suggest program corrections or provide debugging assistance.
- We present Nemo, a prototype automated debugging tool. We verified its effectiveness in repairing errors of omission and identifying root causes of errors of commission on six protocol implementations, of which we discuss four in case studies.

2. BUGGY PROTOCOL MOTIVATION

To motivate our approach, we start with a simple, "buggy" protocol implementation. Figure 1 shows a programmer's first attempt at implementing primary/backup replication [1] in the declarative programming language Dedalus [4]. 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 32–38 of Figure 1. If a payload was marked as acknowledged in table `acked` at the client (*antecedent* predicate `pre`, lines 32–33), then it *must* appear in the `log` of some non-crashed node in the system (*consequent* predicate `post`, lines 34–38). In any run where this is not the case the correctness expectation is violated (details in Section 3.2). The rest of the protocol works as: the primary accepts requests from clients (`request`, lines 3–4) and replicates them to all replicas (`replicate`, lines 17–19), which store them durably in their local state (`log`, lines 22–25).

Unfortunately, the programmer has tried to optimize this protocol for performance. Lines 10–13 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 [28, 21, 2] 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 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 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. [11] 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. Unfortunately, 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*

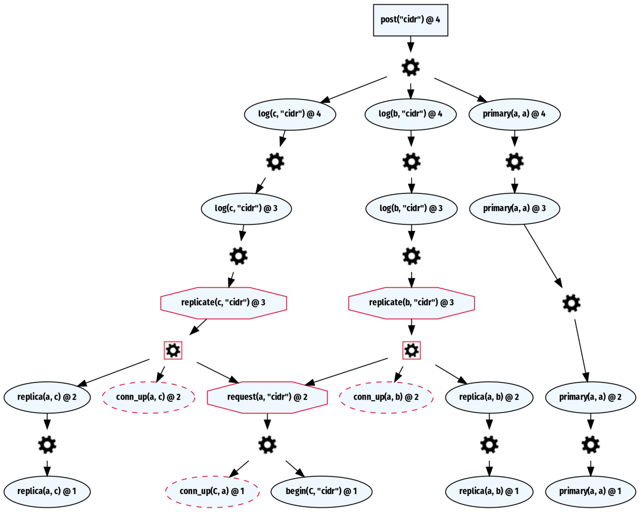


Figure 2: Simplified representation of the consequent provenance graph for a successful run of the Async P/B protocol from Figure 1 in reverse chronological order top to bottom. Consequent predicate `post` (lines 34–38 in Figure 1) is shown as the rectangular vertex at the top. Red-bordered octagonal vertices represent message-passing events (postfixed `@async` in Figure 1). Red-dashed vertices capture network connectivity to the respective other node. The two red-boxed gears hint at computations that might not take place in a failed run, preventing the protocol from establishing the `post` predicate.

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.

3. DEBUGGING 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.

3.1 Assumptions and Terminology

We expect the distributed system under inspection to operate in the *omission fault model*, in which messages may be lost and processes may fail by crashing, but do not exhibit Byzantine behavior. We assume the system to consist of at least two processes that communicate via messages. As input to our strategies, we expect a collection of *provenance graphs* from a series of *runs* of the program. Figure 2 represents one such provenance graph for the consequent of a successful run of the protocol from Figure 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 tests). In practice, we imagine this to be a tight loop, such as the layout visualized in Figure 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.

3.2 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 1) and $\mathcal{C}$ (`post` in Figure 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.

3.3 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 2. 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* ($-$). $A \cap B$ 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

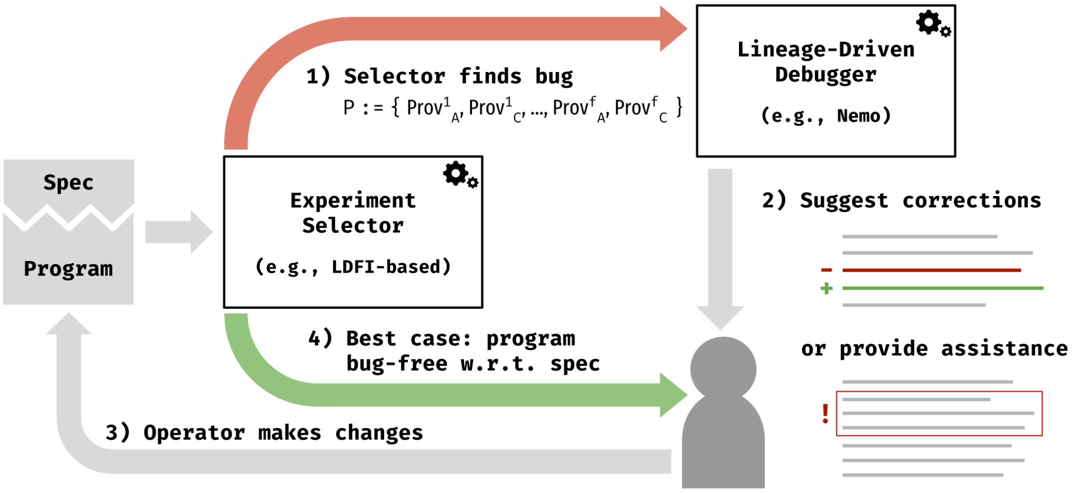


Figure 3: We assume our lineage-driven distributed debugger to be tightly integrated with an experiment selector providing the provenance graphs that form the basis of our analyses. A human operator applies the compiled suggestions.

edges of B are removed. Note, that 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 an outgoing edge. Analogously, $\texttt{roots}(A)$ returns all vertices without an 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.

3.4 Principal Strategies

We now show how our framework expresses common debugging strategies that expose root causes of distributed bugs and assists developers in writing permanent fixes.

3.4.1 *Differential Consequent Provenance*

Differential provenance [11] 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 computation of vertices $\text{Diff}_\mathcal{C}$ over abstract provenance graphs in Figure 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.

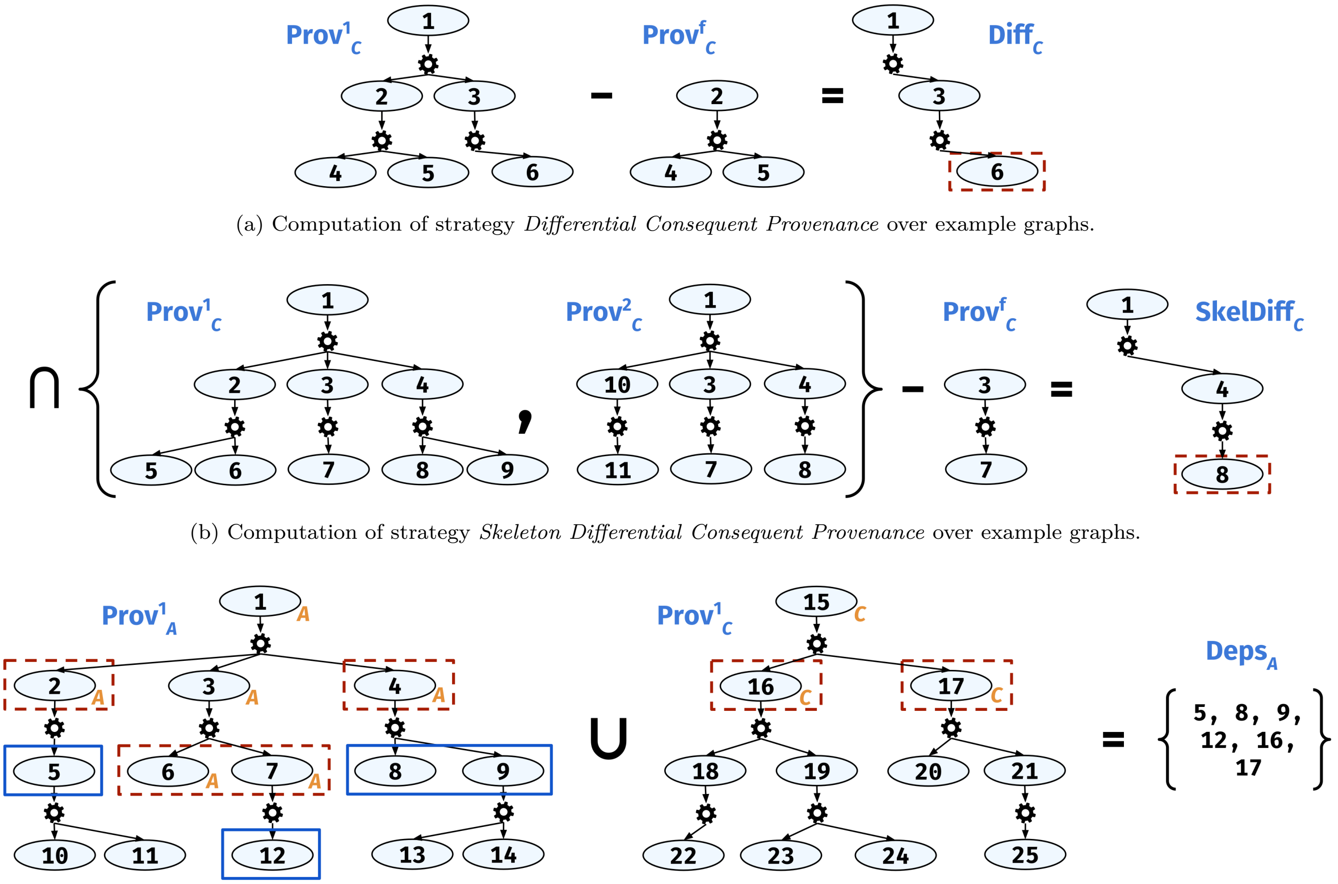
Unfortunately, repairs for errors of omission are not always straightforward, and this approach can be a dead end. Consider again the protocol presented in Section 2. $\text{Diff}_\mathcal{C}$ identifies the rules that failed to fire when messages were dropped between the primary and backups. Focusing 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 that the primary acknowledges the client *too soon*. Differential provenance alone leads us away from this bug.

3.4.2 *Skeleton Differential Consequent Provenance*

When more than one successful run is available, we can take this 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 $\text{SkelDiff}_\mathcal{C}$ with the failed run's consequent provenance:

$$\text{SkelDiff}_\mathcal{C} \coloneqq \texttt{leaves}\Big(\Big(\bigcap_{i=1}^{s} \texttt{normalize}(\text{Prov}^i_\mathcal{C})\Big) - \text{Prov}^f_\mathcal{C}\Big)$$

For intuition, we show a simplified computation of vertices set $\text{SkelDiff}_\mathcal{C}$ in Figure 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



(a) Computation of strategy *Differential Consequent Provenance* over example graphs.

(b) Computation of strategy *Skeleton Differential Consequent Provenance* over example graphs.

(c) Computation of strategy *Corrections Generation* over example graphs.

Figure 4: Exemplary visualization of our three principal strategies for provenance-based debugging. Per strategy, equal vertex numbers identify the same logical event. Red-dashed boxes denote the result of operation `leaves` on a subgraph, blue boxes show the outcome of operation `reachable`. Orange-colored indices at the lower right of a vertex mark that the respective property evaluates to true for that vertex, e.g., vertex with ID 3 is part of $\texttt{prop}_{\mathcal{A}=true}(\text{Prov}^1_{\mathcal{A}})$ in (c).

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 $\text{SkelDiff}_{\mathcal{C}}$ to fire under more fault settings.

3.4.3 Corrections Generation

The two debugging strategies above provide 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 3.4.1, 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. Traces of successful and unsuccessful runs are not enough.

We have one other tool at our disposal, however: the correctness specifications themselves. If we observed a failed run, we know that $\mathcal{A}$ held but $\mathcal{C}$ did not at the end of the execution. Thus, there must have existed a window in the protocol flow during which an injection of the right combination of faults left the protocol no chance to ever establish $\mathcal{C}$. Increasing the number of ways for $\mathcal{C}$ to eventually be established might make the protocol more robust to faults, but it will only postpone the time at which the bug finder injects the right faults that forfeit $\mathcal{C}$ once more.

Going back to our protocol from Figure 1 that is intended 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 prevent $\mathcal{C}$ from being established. No matter how many redundancy measures we add, an intelligent bug finder will always identify at least one run that violates the specification.

One way to rule out the anomalous execution is to ensure that the conditions for establishing $\mathcal{C}$ become conditions for establishing $\mathcal{A}$ as well. Colloquially, instead of repairing the program by making it *easier* to establish $\mathcal{C}$, we can rule out the anomaly more effectively by making it harder to establish $\mathcal{A}$. We can identify the rules leading to $\mathcal{C}$ and generate updated dependencies for $\mathcal{A}$ that precisely include those that cause $\mathcal{C}$ to be established. 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:

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

Omitting details of an actual protocol execution, the updated dependencies set $\mathrm{Deps}_{\mathcal{A}}$ for $\mathcal{A}$ based on exemplary provenance graphs $\mathrm{Prov}^1_{\mathcal{A}}$ and $\mathrm{Prov}^1_{\mathcal{C}}$ is shown in Figure 4c. Distributed correctness properties such as durability are typically global predicates (e.g., "there exists a replica on which the data is stored"), and hence making it possible to test them may require non-trivial changes to the protocol to centralize the required information. If $\mathcal{C}$ 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, $\mathrm{Deps}_{\mathcal{A}}$ will differ such that $\texttt{leaves}(\texttt{prop}_{\mathcal{C}=true}(\mathrm{Prov}^1_{\mathcal{C}}))$ 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 4 that these appear easy enough for developers inexperienced with the protocol to devise and insert into the protocol code.

4. EVALUATION

We validate our debugging strategies using real-world bugs from the TaxDC collection by Leesatapornwongsa et al. [18]. 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 description of steps leading to the bug, and official fix if available. Beginning with the TaxDC corpus, we performed an initial filtering pass that excluded bugs that would be difficult to reproduce given the limitations of the bug finder [2] we used. We then classified the remaining bugs according to root cause, noting for each class whether it is correctable or debuggable with our framework. At the time of writing, we had classified more than half of them. We present the resulting taxonomy, based on the 52 bugs we analyzed in detail, in Section 4.1 and Table 1. We implemented the principal strategies from Section 3.4 in our prototype debugger Nemo [25] and successfully analyzed and fixed six of the bugs from our taxonomy. We present four case studies to demonstrate effectiveness and limitations of Nemo in Section 4.2.

4.1 Bug Taxonomy

In Table 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 our protocol from Figure 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 a basic concept or implementation *bug*.

Of 52 bugs, 24 are potentially repairable by our corrections generation strategy (Table 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 1, column "Assistance"). Further 11 bugs are general mistakes and the effectiveness of our methods depends on the bug at hand. Finally, only for bugs with wrong fallback behavior appear our strategies of no advantage in assistance over conventional debugging.

4.2 Case Studies

We implemented three timing and three logic bugs [25] from Table 1 in Dedalus [4] and submitted them to Molly [23], the reference implementation of lineage-driven fault injection [2]. 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 CA-2083 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 completes. 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

	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	
	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 line that differs in specific config value.
	fallback behavior Actions in response to perceived error are wrong.	✘	✘	5	*Fix:* Rewrite wrong fallback logic or add it at all. *Assistance:* None.
	bug Concept or implementation error.	✘	❍	11	*Fix:* Depends on bug. *Assistance:* Depends on bug.

Table 1: Taxonomy of 52 distributed concurrency bugs from the TaxDC collection [18] and the asynchronous primary/backup protocol from Figure 1. Legend: ✔ = yes, ✘ = no, ❍ = it depends.

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 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. Because the specification references global system state, our repair has to consolidate that state at a single node. 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 making the system indeed 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.

5. RELATED WORK

Provenance [8] provides explanations for *why* and *how* data was computed. The concept originated in the database community [12, 5, 9], but has seen applications in big data analytics [20, 16], operating and storage systems [7, 24], and network diagnostics [11, 29, 30, 10]. *Differential provenance* [11], like Nemo, focuses on the use of provenance as a debugging tool. Reducing representation complexity of provenance graphs, differential provenance reveals the *difference* between a graph of a successful and a failed run, as the root cause of the different behavior is likely to be nearby. We show that differential provenance is a special case of a query in our framework. Wu et al. [29] also explore the use of provenance to guide program repair in the context of software-defined networks. Nemo extends this idea to arbitrary distributed programs, provided that they satisfy the assumptions given in Section 3.

Nemo is complementary to the huge body of work on bug *finding* for distributed and concurrent systems [26, 15, 17, 31]. Our debugger is designed to work in concert with a bug finder; for our prototype, we integrated with an implementation of lineage-driven fault injection (LDFI) [2]. FCatch [19], much like LDFI, specifically focuses on bugs that are triggered by faults such as machine crashes and message loss. Like Nemo, FCatch validates their approach using the TaxDC bug collection. Input minimization techniques (e.g., delta debugging [22] and DEMi [27]) are likewise complementary to our approach, which begins with a reproducible bug.

6. CONCLUDING REMARKS

We built Nemo to explore the use of data provenance as a basis for the distributed debugging tool sets of the future. Although we report nascent work, we showed strong evidence that the question-and-answer process of bug identification and repair can be posed as queries over traces of system execution, identifying root 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. This was no accident: we wanted to explore the *limits* of our approach in a "perfect information" scenario. Generalizing these results to large-scale distributed systems with shallow or non-existent specifications, coarse-grained tracing and logging rather than provenance collection, and a variety of industrial programming languages will require additional research. We argue, that oftentimes integration tests, service-level agreements, and monitoring alerts are already available as a basis for stricter correctness specifications. The generalization of LDFI from the same idealized model to real-world deployments at companies such as Netflix [3] and eBay [14] provides evidence that these problems can be solved. To this end, encoding our provenance framework into a simple, readily available DSL for general-purpose applicability is future work.

7. ACKNOWLEDGMENTS

We would like to express gratitude for comments and suggestions made by Fotis Psallidas and Eugene Wu. Furthermore, we thank the reviewers for their feedback on how to improve this article. This work was supported by NSF grant 1652368 and by gifts from Huawei and Facebook.

References

[1] P. A. Alsberg and J. D. Day. "A Principle for Resilient Sharing of Distributed Resources". In: ICSE '76. IEEE Computer Society Press.

[2] P. Alvaro, J. Rosen, and J. M. Hellerstein. "Lineage-driven Fault Injection". In: SIGMOD '15. ACM.

[3] P. Alvaro et al. "Automating Failure Testing Research at Internet Scale". In: SoCC '16. ACM.

[4] P. Alvaro et al. *Dedalus: Datalog in Time and Space*. Tech. rep. EECS Department, University of California, Berkeley, Dec. 2009. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-173.html.

[5] B. Arab et al. "A Generic Provenance Middleware for Queries, Updates, and Transactions". In: TaPP 2014. USENIX Association.

[6] P. Bailis and K. Kingsbury. "The Network is Reliable". In: *Commun. ACM* 57.9 (2014), pp. 48–55.

[7] A. Bates et al. "Trustworthy Whole-System Provenance for the Linux Kernel". In: USENIX Security 15. USENIX Association.

[8] P. Buneman, S. Khanna, and T. Wang-Chiew. "Why and Where: A Characterization of Data Provenance". In: *ICDT 2001*. Springer Berlin Heidelberg.

[9] A. Chapman and H. V. Jagadish. "Why Not?" In: SIGMOD '09. ACM.

[10] A. Chen et al. "Data Provenance at Internet Scale: Architecture, Experiences, and the Road Ahead". In: CIDR '17.

[11] A. Chen et al. "The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance". In: SIGCOMM '16. ACM.

[12] J. Cheney, L. Chiticariu, and W.-C. Tan. "Provenance in Databases: Why, How, and Where". In: *Foundations and Trends® in Databases* 1.4 (2009), pp. 379–474.

[13] P. Deutsch. *The Eight Fallacies of Distributed Computing*. 1994.

[14] E. Foley. *Peter Alvaro to work with eBay on Lineage-Driven Fault Injection project*. URL: https://www.soe.ucsc.edu/news/peter-alvaro-work-ebay-lineage-driven-fault-injection-project (visited on 09/06/2018).

[15] G. J. Holzmann. "The Model Checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997).

[16] R. Ikeda, H. Park, and J. Widom. "Provenance for Generalized Map and Reduce Workflow". In: CIDR '11.

[17] C. Killian et al. "Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code". In: NSDI '07. USENIX Association.

[18] T. Leesatapornwongsa et al. "TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems". In: ASPLOS '16. ACM.

[19] H. Liu et al. "FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems". In: ASPLOS '18. ACM.

[20] D. Logothetis, S. De, and K. Yocum. "Scalable Lineage Capture for Debugging DISC Analytics". In: SoCC '13. ACM.

[21] B. T. Loo et al. "Declarative Networking: Language, Execution and Optimization". In: SIGMOD '06. ACM.

[22] G. Misherghi and Z. Su. "HDD: Hierarchical Delta Debugging". In: ICSE '06. ACM.

[23] *Molly*. URL: https://github.com/palvaro/molly (visited on 08/24/2018).

[24] K.-K. Muniswamy-Reddy et al. "Provenance-Aware Storage Systems". In: USENIX ATC '06. USENIX Association.

[25] *Nemo*. URL: https://github.com/numbleroot/nemo (visited on 12/17/2018).

[26] C. Newcombe et al. "How Amazon Web Services Uses Formal Methods". In: *Commun. ACM* 58.4 (2015).

[27] C. Scott et al. "Minimizing Faulty Executions of Distributed Systems". In: NSDI '16. USENIX Association.

[28] M. Whittaker et al. "Debugging Distributed Systems with Why-Across-Time Provenance". In: SoCC '18. ACM.

[29] Y. Wu et al. "Automated Bug Removal for Software-Defined Networks". In: NSDI '17. USENIX Association.

[30] Y. Wu et al. "Diagnosing Missing Events in Distributed Systems with Negative Provenance". In: SIGCOMM '14. ACM.

[31] M. Yabandeh et al. "CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems". In: NSDI '09. USENIX Association.