Brief Announcement: Techniques for Programmatically Troubleshooting Distributed Systems

Sam Whitlock
International Computer Science Institute (ICSI)
1947 Center St.
Berkeley, CA 94704
samw@icsi.berkeley.edu

Colin Scott
University of California Berkeley
387 Soda Hall
Berkeley, CA 94720-1776
cs@cs.berkeley.edu

Scott Shenker
ICSI & University of California Berkeley
387 Soda Hall
Berkeley, CA 94720-1776
shenker@icsi.berkeley.edu

ABSTRACT

The distributed systems research community has developed many provably correct algorithms and abstractions that are in wide use. However, practical implementations of distributed systems often contain many bugs, and practitioners spend much of their time troubleshooting these bugs. In this paper we present an algorithm, retrospective causal inference, to ease the burden of troubleshooting. We end by enumerating several open research problems related to the troubleshooting process.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*distributed debugging, debugging aids, tracing*

General Terms

Algorithms, Theory

Keywords

troubleshooting; automation; tools

1. INTRODUCTION

Despite a wealth of abstractions and provably correct algorithms developed by the distributed systems research community, practical implementations of even simple distributed systems often contain bugs. Finding and fixing the causes of these bugs is a time-consuming task. For example, a 2006 survey found that software developers at Microsoft spend 49% of their time troubleshooting bugs [1]. The same study found that 70% of the reported concurrency bugs take days to months to fix, and 74% of respondents considered bug reproducibility hard or very hard.

The *de facto* method for troubleshooting is painstaking manual analysis of runtime logs. Such manual troubleshooting is hindered by the large number of inputs to distributed systems; troubleshooters find little immediate use from traces containing many inputs prior to a fault, since they are forced to manually filter extraneous inputs before they can start fruitfully examining the source of the errant behavior. It is no surprise that when asked to describe their ideal tool, most practitioners said "automated troubleshooting" [9].

We have developed an algorithm, retrospective causal inference, as a step towards automated troubleshooting. Given a trace of causally-ordered events that leads to a bug, retrospective causal inference finds a locally minimal subsequence of the trace that is sufficient for reproducing the bug. This smaller set of events provides developers a better understanding of how the bug in their code was triggered.

We have applied retrospective causal inference [4] to one type of distributed system: control software for software-defined networks. Our initial experiments have been highly promising: of five bugs discovered in a five day investigation, retrospective causal inference reduced the size of the input trace to 18 events in the worst case and 2 events in the best case. In this brief announcement we describe retrospective causal inferenceand enumerate several open research problems related to the troubleshooting process.

```
procedure REPLAY(subsequence)
 for e_i in subsequence
   if e_i is an internal event
    and e_i is not marked absent :
      then { Δ ← |e_i.time − e_i-1.time| + ε
             wait up to Δ seconds for e_i
             if e_i did not occur :
               then mark e_i as absent
   else if e_i is an input :
      then { if a successor of e_i occurred :
               comment: waited too long
               then return REPLAY(subsequence)
               else inject e_i

comment: See Figure 2 for invocation
```

Figure 1: `Replay` is responsible for replaying subsequences of events chosen by delta debugging (Figure 2) and determining if the bug reappears.

2. APPROACH

One well-known method for finding event traces that lead to bugs is symbolic execution [3]: given source code as input, symbolic ex-

PODC'13, July 22–24, 2013, Montréal, Québec, Canada.

ecution builds a model of the distributed state machine. With a model of the distributed state machine in hand, finding and minimizing errant event traces is fairly straightforward. Unfortunately, building this model from source code involves enumerating an exponential number of code paths. Although formal methods can, in theory, locate and minimize errant behavior, it quickly becomes an impractical option for real systems.

2.1 Minimized Event Sequences

Practitioners instead typically rely on execution logs to help them debug their distributed systems. Execution logs represent a particular subpath through the distributed state machine (where each event in the log represents a state transition) that is known to trigger a bug.[1]

Execution logs can be quite large, and it is time-consuming to analyze them by hand. It is not apparent from the logs which events are relevant and which are extraneous, leaving the developers to use their judgment about which code paths led to the errant behavior.

Deterministic replay systems allow troubleshooters to reproduce system executions, but merely replaying the execution does not immediately aid in deducing which transitions were causally related to the bug and which were not. Practitioners need a small sequence of transitions that triggers the bug. By examining each of the state transitions in such a sequence, they can get a better understanding of the code path that contains the root cause. Thus, given a bug-inducing execution log, it would be highly desirable to automatically find the *minimal causal sequence* (MCS): a subsequence of the trace that leads the state machine to a bug state, and has the additional property that if any of the transitions are removed from the sequence, the bug state is not reached. Finding minimal causal sequences is the goal of our work.

2.2 Replay, Pruning, and the Functional Equivalence of Events

The software engineering community has explored several search algorithms [6,8] for minimizing test cases. We focus on a particular algorithm suited to our goal: delta debugging [8]. Given a single input (*e.g.* an HTML page) for a non-distributed program (*e.g.* Firefox), delta debugging repeatedly runs the program on subsets of the input until it finds a minimal subset (*e.g.* a single tag) that is sufficient for triggering a known bug.

Delta debugging has not yet been applied to distributed systems. Doing so is complicated because the inputs to distributed systems are spread across time and across multiple processes, rather than being injected at a single point in time into a single process. This substantially complicates the task of testing whether a subsequence chosen by delta debugging contains the minimal causal sequence.

Elsewhere, we describe a system for replaying event subsequences chosen by delta debugging [4]. Here we focus on the algorithmic challenges posed by our system. When we replay a subsequence chosen by delta debugging, we must ensure that causality is maintained. Specifically, to reliably reproduce the original bug we need to inject each input event e at exactly the point when all other events, both internal to the system (such as messages sent between nodes or internal state changes) and external to the system (such as node crashes that we inject), that precede it in the happens-before relation ($\{i \mid i \rightarrow e\}$) from the original execution have occurred [5].

At first glance, it seems that allowing delta debugging to alter the history of the log will prevent us from being able to maintain causality during replay; if we diverge at all from the original event trace we may find ourselves on a different path through the state machine, unable to reason about the original happens-before constraints. Consider for example that the sequence numbers of the messages passed throughout the system may change if we prune a single event at the beginning of the trace. Once on a diverged path, it is unclear whether the original bug will still be triggered.

Our approach to coping with divergence has been to apply heuristics to allow us to maintain causality as best we can. First, we explicitly disallow delta debugging from subdividing the trace in a way that leaves an invalid input sequence. We accomplish this by telling delta debugging to only remove atomic groups of inputs. For example, if we prune a controller failure event, we make sure to prune the controller's subsequent recovery event. This approach currently depends on our domain knowledge of the semantics of input events.

Next we need to cope with the fact that the syntax of internal events may change subtly after pruning inputs. We observe that many events are *functionally equivalent*, in the sense that they have the same effect on the state of the system with respect to triggering the bug (despite syntactic differences). For example, it is unlikely that the code responsible for incrementing the sequence number of messages is related to a buggy replication algorithm, meaning that we can often safely ignore the sequence numbers of messages. By disregarding irrelevant state, we draw an equivalence relation between the events across divergent runs, allowing us to compare executions generated by different subsequences of the original event trace. In this way we can maintain causality of events in the same equivalence classes.[2]

When comparing a pruned execution history to the original, there are two other subtleties we need to consider: some events from the original execution may be absent, and other events may be entirely new. We cope with absent events by timing out after some duration if they do not occur. We currently allow new events—internal events that are not functionally equivalent to any events observed in the original trace—to simply occur, and do not use them to dictate the timing of external events.

2.3 Retrospective Causal Inference

We have developed an algorithm, retrospective causal inference, that combines these heuristics to find minimal causal subsequences of event traces. We show pseudocode for retrospective causal inference in Figure 2. Using our replay system [4], we have found and minimized several buggy traces in open source distributed systems.

3. OPEN PROBLEMS

The algorithm we described in the previous section has worked well in practice, but it leaves open some questions that, if addressed, may yield a more principled approach to troubleshooting distributed systems.

How should new events be handled? New events ultimately leave open multiple possibilities for where we should inject the next input. Consider the following case: if i_2 and i_3 are internal events observed during replay that are both in the same equivalence class as a single event i_1 from the original run, we could inject the next input after i_2 or after i_3.

Exploring both possibilities would incur exponential runtime. Our approach to dealing with new events, ignoring them, is a heuristic. We believe that there may be more principled approaches that are still tractable for large real-world systems.

[1] We have shown elsewhere [4] how these event traces can be obtained.

[2] Incidentally, ignoring extraneous message fields is similar to how practitioners examine event logs by hand: they intuitively disregard certain information they deem to be irrelevant.

Figure 2: Delta Debugging Algorithm From [7]

Input: $T_{\text{✘}}$ s.t. $T_{\text{✘}}$ is a trace and $\mathit{Replay}(T_{\text{✘}}) = \text{✘}$. Output: $T'_{\text{✘}} = \mathit{ddmin}(T_{\text{✘}})$ s.t. $T'_{\text{✘}} \subseteq T_{\text{✘}}$, $\mathit{Replay}(T'_{\text{✘}}) = \text{✘}$, and $T'_{\text{✘}}$ is minimal.

$$\mathit{ddmin}(T_{\text{✘}}) = \mathit{ddmin}_2(T_{\text{✘}}, \emptyset) \quad \text{where}$$

$$\mathit{ddmin}_2(T'_{\text{✘}}, R) = \begin{cases} T'_{\text{✘}} & \text{if } |T'_{\text{✘}}| = 1 \text{ (“base case”)} \\ \mathit{ddmin}_2(T_1, R) & \text{else if } \mathit{Replay}(T_1 \cup R) = \text{✘} \text{ (“in } T_1\text{”)} \\ \mathit{ddmin}_2(T_2, R) & \text{else if } \mathit{Replay}(T_2 \cup R) = \text{✘} \text{ (“in } T_2\text{”)} \\ \mathit{ddmin}_2(T_1, T_2 \cup R) \cup \mathit{ddmin}_2(T_2, T_1 \cup R) & \text{otherwise (“interference”)} \end{cases}$$

where $\mathit{Replay}(T)$ denotes the state of the system after executing the trace T, ✘ denotes a correctness violation, $T_1 \subset T'_{\text{✘}}$, $T_2 \subset T'_{\text{✘}}$, $T_1 \cup T_2 = T'_{\text{✘}}$, $T_1 \cap T_2 = \emptyset$, and $|T_1| \approx |T_2| \approx |T'_{\text{✘}}|/2$ hold.

Is there a better definition of functional equivalence? Our current definition of functional equivalence is based on guess-work and domain knowledge: we intuitively disregard certain pieces of information in each internal event because we know that they typically do not determine whether the bug appears. We believe that formulating functional equivalence in terms of knowledge states of the distributed system [2] will yield a more principled approach. If retrospective causal inference has insight into what pieces of knowledge the nodes of a distributed system act upon, it can make stronger statements about what parts of each message or internal state change are capable of affecting the actions of a given node. By comparing the internal events across runs based on what pieces of information actually affect the end state, we can draw stronger functional equivalencies between events.

What types of systems are most amenable to troubleshooting? Suppose you are troubleshooting a bug in a system that takes a small number of inputs and performs a long series of computations after receiving the inputs. If you prune a single input event, every successive intermediate state, including the final state, diverges from the states in the original execution. In this scenario, the minimal causal sequence will often be the original event sequence in its entirety, and the minimization provided by retrospective causal inference would have no value.

We conjecture that retrospective causal inference operates best on systems that have ‘quiescent’ state machines, where each input event only triggers a small number of internal state transitions. We successfully tested retrospective causal inference on control-plane systems (software-defined network controllers), which tend to have this property. It is unclear whether other systems would be equally amenable to retrospective causal inference.

What types of bugs does this technique perform poorly well on? Even with a MCS in hand, troubleshooters need to match the MCS to the errant code path that ultimately triggers the bug. Retrospective causal inference helps by making the execution trace easy to understand, but the programmer may not easily be able to match the sequence to a code path. Consider for example a bug that is triggered the 27^{th} time the letter ‘a’ appears in the sequence of events. There are many different event subsequences that can trigger such a bug. Due to the difficulty of distributed replay, it is very possible that retrospective causal inference will return different bug-triggering sequences on successive invocations. It would require a great deal of insight for a practitioner to deduce a pattern they all share.

We are not certain what types of bugs retrospective causal inference is most useful for. In our experience, bugs that are triggered without prior complicated event sequences provide short, insightful MCSes. For example, if a system does not support a particular type of input event (*e.g.*virtual machine migration), then the first occurrence of such an event in any sequence of input events will trigger the bug; in such a case, retrospective causal inference will return the single event.

4. CONCLUSION

Developers of distributed systems must be mindful of the minute details of computer systems. The troubleshooting tools available for single-process systems are rarely applicable to distributed systems, leaving practitioners with few viable troubleshooting options when they observe errant behavior in their distributed systems. Developers consequently spend much more of their time and effort troubleshooting bugs. We believe that a principled approach to automated troubleshooting is possible, and we have taken a first step here by presenting an algorithm for automatically minimizing errant event traces. We hope that the distributed systems community will further investigate the theoretical issues behind systematic troubleshooting.

5. REFERENCES

[1] P. Godefroid and N. Nagappan. Concurrency at Microsoft - An Exploratory Survey. CAV ’08.

[2] J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. JACM ’90.

[3] J. C. King. Symbolic Execution and Program Testing. CACM ’76.

[4] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker. How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software. Technical Report UCB/EECS-2013-8, University of California, Berkeley, ’13.

[5] G. Tel. *Introduction to Distributed Algorithms*. Thm. 2.21. Cambridge University Press, 2000.

[6] A. Whitaker, R. Cox, and S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. SOSP ’04.

[7] A. Zeller. Yesterday, my program worked. Today, it does not. Why? ESEC/FSE ’99.

[8] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE TSE ’02.

[9] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A Survey on Network Troubleshooting. Technical Report TR12-HPNG-061012, Stanford University ’12.