Application-Integrated Record-Replay of Distributed Systems

Narek Galstyan

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-4
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-4.html

January 12, 2024

Application-Integrated Record-Replay of Distributed Systems

Abstract

This report reviews bug catalogs and debugging systems designed for distributed systems. It tries to find common patterns in distributed systems bugs, highlights the characteristics necessary in a debugging system to identify these bugs in distributed systems, and proposes the Application-Integrated Record-Replay (aiRR) system for addressing classes of these bugs.

aiRR is designed specifically for distributed systems. aiRR integrates the recording into the distributed system and leverages this integration to reduce the overhead of recording in the application. To have low overhead, our approach avoids reducing application-level concurrency and avoids recording application-level data that is not necessary for replay.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Distributed Systems, Debugging, Failure Reproduction, Bug Catalog

1 Introduction

Distributed systems are notorious for containing subtle bugs that manifest only as failures in production [17, 20, 27]. Despite the vast literature of known testing and verification techniques [13, 35], distributed systems bugs still often make it to production.

Even widely production-deployed systems, such as Etcd [8] and Zookeeper [9], get frequent bug reports from production runs. These systems have been heavily tested, both by their authors and by all the companies that deploy them in their infrastructure. But the complexity of these systems means that there are still many edge cases that are not handled properly and could become (and do become [2]) the trigger for a production failure.

Reproducing the production failure in a controlled environment for understanding the bug is a **critical** step in post-mortem debugging. Because of the inherent complexity of distributed systems, many bugs, particularly those that made it to production, are hard to trigger and reproduce. It is also often impossible to make any progress in bug fixing without first reproducing the bug.

MS Technical Report, Dec 10, 2023, Berkeley, CA, USA
2023.

Reproducing the production failure is also a **time-consuming** step of debugging. Many publicly reported bugs in Zookeeper and Etcd took months to resolve after the initial bug report was submitted. According to some open-source issue board analyses [41], HDFS, HBase, and Zookeeper developers spend an average of 70% of bug resolution time reproducing the bug. For example, it took 3.5 months from its first report for a recent Etcd bug to be reproduced. The fix was merged shortly after it was reproduced.

Existing post-mortem debugging tools cannot reproduce distributed systems failures in a controlled setting. As a result, although they are helpful in debugging performance issues, they do not help in debugging one-off failures in distributed systems. Selective logging systems such as Dapper [34] log information about a tiny minority (about 0.1%) of requests, so information about the fault may not even be in the recorded trace. Even when it is in the recorded trace, using the logs to make sense of the complex control flow of the distributed systems is often impossible. Verified distributed systems [13] promise to eliminate bugs by construction, but are often difficult to apply to production systems that are more tightly integrated into an application and business logic. Off-the-shelf record-replay systems [4, 5, 25, 26, 36], have too much overhead to be enabled on production distributed systems.

In this work, we aim to improve the debuggability of production failures in currently deployed distributed systems. We first survey various approaches for debugging and their suitability for distributed systems. We discuss the applicability of these systems to concrete bugs from bug catalogs in the literature.

Motivated by our findings from debugging and bug catalog literature, we propose aiRR– an application-integrated record-replay system that has many of the advantages of state-of-the-art record-replay systems but takes advantage of the particular structure and assumptions of typical production distributed systems to be low overhead and, therefore, allows production deployment of recording.

We notice that record-replay systems have two key sources of overhead: 1) reduced concurrency and 2) the recording of large app-level buffers. State-of-the-art recording systems reduce the overhead of recording concurrency primitives [25] or avoid persisting recorded data [21] to improve performance. The resulting overhead and the setting of not persisting full recorded trace are not acceptable for debugging many production distributed systems failures.

In aiRR design, we aim to reduce the overhead from the two sources above, by specializing record-replay system to the distributed system under recording. We take advantage of two key aspects of distributed systems that are not applicable to general applications and that allow us avoid these sources of overheads.
First, we notice that in distributed systems not all instances of concurrency primitives (mutexes, atomics) need to be recorded for successful replay of a wide range of bugs. We categorize uses of concurrency primitives according to their purpose within the application and only record the instances of primitives that are necessary for replay.
Second, we notice that distributed systems typically use one of the few RPC communication libraries. In aiRR, we provide interfaces that integrate with these communication libraries and allow the application developer to selectively avoid recording costly application-level buffers that are not necessary for replay.

The rest of this paper is organized as follows: we discuss the related work on debugging and bug reproduction in Section 3. We describe the prior work in terms of efficiency, effectiveness, and accuracy and note that record-replay systems are getting close to being a good solution for debugging distributed systems in production.

We then discuss a framework from the past bug catalog literature for reasoning about distributed systems bugs in Section 4. We discuss various bug catalogs in terms of the framework, and create a bug map, positioning bugs according to what kind and how much information is necessary to reproduce them. We conclude the section with a discussion of the effectiveness of past debugging systems (from Section 3) in debugging various regions of the bug map.

In Section 5 we give an overview of our proposed distributed system record-replay system. We describe the implementation and API of aiRR in Section 6 and evaluate it in Section 7 via microbenchmarks, production bug reproduction examples, and recording overhead measurements.

2 System Model

Throughout this paper, we model distributed systems as independent nodes that run some computation and can fail independently. We model each individual node as an IO automaton [22] that communicates via sending and receiving messages and maintains an internal state that can change in response to incoming messages.

Node faults and restarts are a special sub-category of internal state changes. The distinction is relevant for the discussion in Section 4.5 where we discuss the record-replay overheads of a system that follows this system model.

Each individual node communicates with other nodes as well as with its environment. So, random numbers, timer values and other external environmental parameters for a node are modeled as messages sent from the environment to the node.
An event in this model is one of the set {message arrival/send, local computation, fault/reboot} elements.

We use an IO-automaton-like model in this work because many algorithms for distributed systems such as Raft [28], Zab [14], IPFS [6] are modeled in the IO automaton model in literature. The real world distributed systems implementing these algorithms such as Etcd [8] and Zookeeper [9] typically follow a similar model in their implementation code.

We leverage this model in Section 4 to position bugs from various bug catalogs on a common map, and use the model's assumptions in aiRR design to reduce recording overhead.

3 Related Work

We start by talking about the unique challenges that arise in distributed systems, and then we lay out the key properties that developers trade between when choosing a debugging approach. We talk about four major categories of distributed debugging tools characterizing their point in the trade-off space, as well as their strengths and weaknesses.

There are three key properties to consider when designing a debugging system [43]. **Efficiency** is the overhead on the system to improve debugging. This is particularly important for debugging in production. **Effectiveness** is the power of the system to help developers diagnose a wide variety of problems. **Accuracy** is the fidelity with which a debugging system enables reproducing the execution state of the failure. Now we describe where the common debugging approaches land on this trade-off space.

Logging systems allow developers to trace execution paths, as well as to understand the value of data by placing log statements in the codebase. These log statements add extra cycles to print messages to disk as well as pass values in adding a good amount of overhead. Their effectiveness and accuracy depends on either the developer or system's ability to: place log statements in places that disambiguate execution paths, and choose values to print that are relevant to the failure. These properties are summarized in Table 1.

Program analysis aims to use a semantic understanding of the program to enable reproducibility of failures. Whether this scheme is efficient or not depends on the approach used (e.g. offline/online). The effectiveness and accuracy similarly depend on the specific instantiation.

Formal methods aim to provably eliminate bugs from distributed systems alltogether. More lightweight formal methods try to relax the proof requirement and expand set of systems the methods are applicable to. Although these methods work well for verifying the correctness of protocols, they are less aplicable to real-world distributed systems implementations because of the significantly larger set of features that need to be modeled in the formal model.

	Efficient	Effective	Accurate
Logging	(-) Adds CPU cycles and data marshalling	(+/-) Depends on where logs are placed	(+/-) Depends on where logs are placed and what values are logged
Program Analysis	(+/-) Can be done online/offline	(+/-) Depends on the scheme used	(+/-) Depends on the scheme used
Formal Methods	(+) Done offline	(-) Limited by expressiveness of formalism	(-) Leaves gap b/w formalism and implementation
Record/Replay	(-) Requires expensive logging	(+) Can record any kind of bug	(+) Faithfully represents execution

Table 1: E-E-A trade-off for various approaches to distributed system debugging.

Lastly, **record/replay** schemes capture the statements in the execution path that led to a failure. Since the statements are not known a priori these statements are captured during execution harming efficiency. However, since they capture exactly the statements running during the failure they can capture all bugs and accurately capture those bugs.

3.1 Logging Systems

Logging systems as previously described place statements across the code base to provide developers with information about the code's execution. When considering systems that use logging as a debugging tool, there are three relevant questions we must answer:

(1) Where should logs be placed? The location of the logs has an influence on the ability of logs to discern different execution paths.
(2) What variables should be logged? Variable values can provide important information to developers such as whether program invariants are being withheld, or performance characteristics of the system.
(3) When do you make these decisions? Answering the previous two questions at development time prevents the developer's finding bugs from tailoring logging to their specific failures.

3.1.1 ***PivotTracing:*** PivotTracing [24] aims to help developers debug distributed systems bugs by providing them the ability to ask system-level questions and automatically translate these system-level questions into runtime logs. With what they call "happened-before join", Pivot Tracing gives users, at runtime, the ability to define arbitrary metrics at one point of the system, while being able to select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries.

This can be helpful for debugging many of the bugs in all of the bug map on Figure 1. However, since this is a runtime query engine, the bug must be reliably triggerable so the developer can continually trigger it, collect different system level log data and trace down the root cause of the bug.

3.1.2 ***Dapper:*** Dapper [33] is a tracing system developed by Google for general use for their applications. This system focuses on a design that leverages the cooperative nature of Google's infrastructure. The first requirement they have is ubiquitous deployment. The larger the coverage of a tracing system the more informative. This might be impractical in other scenarios where a distributed system might be making use of external libraries of which it has no control, but is sensible for Google. The next requirement is continuous monitoring. Many papers have highlighted that bug reproduction takes up a considerable amount of resolution time (as high as 70% [41]), so instead Dapper was designed to always be running so that reproduction could be fast. And while they want continuous monitoring they also want low overhead since Google's production applications are heavily optimized to improve user experience. The last requirement is developer transparency. In the common case developers should not have to perform any instrumentation to receive logging information from Dapper.

Dapper performs tracing at the request level. In order to tune the overhead taken up by the system Dapper performs adaptive sampling. For every request it chooses probabilistically whether the request will be a sampled request. Instead of requiring developers to add instrumentation points in their

code Dapper adds instrumentation points to common google libraries. They modify their thread library to attach trace context to thread storage, their async library to add trace context to callbacks/invocations, and their RPC library to transmit trace IDs from client to server. Traces in dapper are made up of spans that form a tree. Spans represent individual units of work and contain timing data, and annotations optionally added by applications.

3.1.3 ***Log20:*** Log20 [42] focuses on determining where log statements should be placed. They outline multiple problems with current log print statement (LPS) practices that they discovered by studying revision history of multiple codebases. First they found that LPSes are often added only after a failure occurs, which means that the developer was not able to debug the issue as it happened. Second, they found that it's hard to predict how useful and expensive adding an LPS will be, pointing out that many revisions only modify the verbosity level (e.g. from ERROR to INFO). Lastly, they find that the scalar nature of verbosity levels is difficult for developers to reason about.

Log20 makes it easier to use logs for debugging distributed systems, but it still has many of the issues discussed above for Dapper.

3.2 Program Analysis Systems

Program analysis uses a semantic understanding of the program to enable reproducibility of failures. It's an approach that offers flexibility to the developer to navigate the trade-off space as they see fit and optimize for a particular use case. They are free to perform offline work to reduce overhead, but could benefit from data gathered online.

3.2.1 ***Failure Sketching:*** The authors propose Gist [15] that relies on hardware watchpoints and hardware features for extracting program control flow efficiently to record what they call "failure scetch" of a failure. A failure sketch concisely lists the statements that led to the failure and describes the delta between a failed and successful execution. Gist automatically produces failure sketches for a given failure using hybrid static-dynamic analysis. It uses program slicing which determines the set of statements that affect the values at a given point in the program.

3.2.2 ***Execution Reconstruction:*** The author's of Execution Reconstruction [43] believe that most debugging systems over commit to some of the properties in the efficiency-effectiveness-accuracy trade-off and thus unnecessarily lose out on some of the other properties. This project aims to use symbolic execution but only to reconstruct the state required to elicit a single control path, one that induced a production failure. This is called shepherded-symbolic execution. However, they state (based on evaluation) that shepherded-symbolic execution is not sufficient to improve analysis times due to the progressive complexity of input constraints.

The insight of this project is to record the minimal set of data values that enable low overhead symbolic execution while maintaining accuracy. They call this process Key Data Value Selection and rely on some hardware support. This involves constructing a constraint graph and using it to find data values that are found in complex dependency chains. In particular they look for long chains of symbolic writes and look for accesses to large symbolic memory objects. Key Data Value Selection is done iteratively. At each step it uses the current data to determine which data values will help to reduce the execution time of symbolic execution by choosing values that fit the constraints output from symbolic execution.

3.2.3 ***Pensieve:*** Pensieve [41] recognizes that developers don't debug failures by reconstructing the full execution path that led to the failure. Developers skip most of the code and find statements that have a causal relationship with the failure. For example, to understand why a variable has a certain value a programmer would instantly jump to the place where it's defined, not scan the entire path. And to deal with ambiguities where the variable could have been defined in 1 of many places a developer uses print statements to determine the relevant place.

Pensieve produces a partial trace with a set of events that occurred during failure execution rather than a full execution. The paper defines the event-chaining algorithm designed to produce these partial traces. There are condition events representing conditions that hold at a location, location events that represent execution having reached a location, invocation events that represent methods being called, and output events representing messages being printed. The algorithm starts with the output events that represent the failure execution and works backward to generate the trace. For each event Pensieve uses data flow analysis to find an event that explains it, and that event is also analyzed to find some parent event. It includes procedural analysis to ignore loop iterations that have no bearing on events, as well as a process to determine how to handle multiple possibilities to explain an event.

3.3 Formal Methods

Approaches discussed so far are reactionary - they assume bugs have already made it to production software and discuss ways to find and fix them. Given that the methods discussed so far have not eliminated all bugs from production systems already, you might wonder: Is it possible to prevent bugs from entering production systems in the first place, so that there is no longer a need to debug?

Fueled by this desire, researchers have attempted to formally verify (or, at least, comprehensively validate) the correctness of distributed systems before putting them into production. This is akin to a Java compiler ensuring that each function call in a program passes the expected number of correctly-typed arguments, so that no bugs can arise at runtime from argument mismatch.

Because formal methods require a very precise model, they are typically applied to mathematical descriptions of algorithms, as opposed to their real-world implementations. This allows specifying security invariants in the same mathematical model and carrying out proofs about the system. Ivy [30] is an example of such a system. Though the authors originally used Ivy to prove invariants in simple procotols such as Chord and distributed locking schemes, it was later extended [29] to prove safty invariants of six Paxos variants.

More recent practical work on the Rabia state-machine replication framework used Ivy to verify its core protocol [31] but does not use any formal methods for the actual implementation of the system.

Applying formal methods to complex, practical distributed systems has two bottlenecks. First, it is harder to model more complex systems and second, it is computationally more expensive to carry out proof computations on more complex systems. IronFleet [13] and Verdi [37] propose several techniques to divide a system into simpler components, each of which can be modeled and verified in isolation. Several recent works [12, 16, 23, 38, 39] have made progress in reducing the proof burden in complex system models.

A significant gap still remains between the verified protocols and their practical implementations. This disconnect highlights a broader issue in the field: While the verification of protocols is advancing, translating these verified protocols into efficient, real-world systems remains a challenge.

3.4 Record/Replay Systems

Record/replay systems record events as they happen on a system in order to replay them later if a bug arises. This enables developers to capture bugs that happen infrequently and diagnose the root cause. These systems are often heavyweight since they attempt to faithfully capture the execution environment. Reducing this overhead is a big challenge in this space.

3.4.1 ***RR*** This paper [26] introduces RR, a practical system for recording and replaying program executions, originally designed by Mozilla engineers to debug the browser. It is designed to be used in a wide array of applications like reverse-execution debugging and black box forensic analysis of failures in deployed systems. R maximizes deployability by working with unmodified user-space applications, stock Linux kernels, compilers, and language runtimes, without requiring pervasive code instrumentation or special privileges. It records all OS-level and thread-scheduling level nondeterminism by continually monitoring the application under recording.

RR has lower overhead than fully ptrace-based approaches. This is largely due to the novel in-process system-call and thread nondeterminism interception technique, which dramatically reduces context switches during system call monitoring. RR still has 2-5x overhead on typical distributed system applications, so it is not practical to run in production.

3.4.2 ***Lightweight RR [18]*** recognizes the usefulness of record-replay for debugging and notes that with additional assumptions overhead of recording could be lowered. Lightweight RR is designed to record channel-based single-node (not distributed) applications written in Rust. It instruments the channel primitive to record message ordering and provides exact replay of message of this order.

Despite the application-level instrumentation, the system still incurs over 2x overhead. Although the authors do not mention the sources of the overhead, our hypothesis is that it is the overhead of recording large application-level buffers.

3.4.3 ***Castor [25]*** Castor is a record/replay system for multi-core applications designed to provide consistently low and predictable overheads so it can run recording on production applications.

Castor is written for FreeBSD and takes advantage of FreeBSD's standard library HAL annotations to generate most of the logic necessary to record OS nondeterminism.

Castor does not require modifying source code thanks to compiler instrumentation for non-deterministic events.

Although we were unable to run Castor ourselves, we ran microbenchmarks that model Castor on the distributed systems we evaluated in this work. Our microbenchmarks indicate that Castor in on-disk trace recording mode (necessary for long-running distributed systems where a fault may not immediately manifest itself) would have prohibitive overhead when recording applications which make heavy use of coordination primitives (locks, atomics) or which send or receive large application level buffers (distributed kv-stores and databases with large values).

3.4.4 ***Debug determinism*** In Debug Determinism [40] the authors recognize that current works have focused on reducing the overhead of record/replay at the expense of effectiveness. They argue that record/replay systems should provide debug determinism, which means that the replay system provides an execution that produces the same failure as well as the same root cause of failure.

They then attempt to formalize a metric that displays the power of a record/replay system. They define debugging fidelity (DF) as the ability of a system to accurately reproduce

the root cause and failure. If the system does not reproduce a failure $DF = 0$, if the system reproduces the root cause $DF = 1$, otherwise $DF = \frac{1}{n}$ where n is the number of possible root causes for the failure that was replayed. Then they define the debugging efficiency (DE) as the time the original execution took divided by the time the system takes to reproduce a failure. They then define the utility of a debugging system as $DF * DE$.

4 Bug Catalogs

In this section we explore previously identified and patched distributed systems bugs to understand better the bugs that commonly get to production in deployed distributed systems. We summarize the distributed system bug taxonomy of TaxDC and use that as a framework to look at other bugs from bug collections of FlyMC, DEMi and curated bugs from Etcd. We summarize the discussion in a bug map in Figure 1.

Lastly, we discuss the trade-offs of using various debugging systems presented in Related Work to address bugs in these catalogs.

4.1 Ordering and Trigger Constraints

Distributed systems bugs are often very involved – they require long sequences of low-probability events to occur and require those events to occur in some specific order for the bug to manifest. To better understand such bugs, in this section we adapt and extend a framework from TaxDC bug catalog which looks at bugs from two distinct classes of constraints.

(1) Ordering constraints: For every bug, they identify the smallest set $\mathbb{E}$ of concurrent events necessary for the manifestation of the bug. Elements of $\mathbb{E}$ can be any type of event specified in the set above. Note that here we do not care about the likelihood of events we put in $\mathbb{E}$ and only care about their ordering, taking their existence as a prior.

(2) Trigger constraints: For the events in $\mathbb{E}$ to occur in some order, regardless of the actual order, certain triggers must occur in the system (e.g., node crashes, view change/leader election, packet delay, etc.). Trigger constraints are the requirements on the occurrence and order of such triggers to obtain the events necessary for $\mathbb{E}$. Here, we do not care about ordering the events in $\mathbb{E}$ but merely identify triggers necessary to cause them in an arbitrary order.

Ordering Constraints: Though most bugs have requirements on both fronts, evaluating the bug along each dimension separately is helpful, as it allows us to create more general bug categories. TaxDC further categorizes each constraint class by observing certain commonalities in the bug reports in their dataset. Overall, 64% of TaxDC bugs are triggered by a single ordering constraint - a single message delivery constrained to arrive at a particular order with respect to other messages or computation is enough to trigger the bug, and changing the delivery order of that single message makes the bug not manifest. Note that the ordering constraints above refer to ordering all kinds of events - messages w.r.t. other messages, messages w.r.t. faults, or other internal state transitions. TaxDC does not categorize ordering constraints of order-dependent DC bugs in this manner. However, they mention the nature of the ordering constraint in several of the bugs they describe in detail. With that information, we can further categorize ordering constraints into the following:

`message-message`,
`fault-message`, `compute-message`,
`compute-fault`, `compute-compute`

where `message`, `compute` and `fault` are as defined in our sytem model.

Each ordering constraint specifies the ordering of two events necessary for a bug to manifest. These categories divide the constraints in terms of event types. If the constraint is on the order of two message deliveries (e.g., receive the Heartbeat before ReadIndex in a raft/consensus system) or on the order of a message delivery and a send (e.g. send VoteRequest then receive a Heartbeat), then we have a message-message ordering constraint to reproduce the bug. If the constraint is on a message delivery with respect to a fault or a restart (e.g., MsgAck is sent before a crash), then we have a `msg-fault` ordering constraint. The other categories are defined similarly. We will use these finer-grained ordering constraint categories to map bugs from various studies on a 2D plane. Later, we will use these categories to explain overhead-coverage trade-offs of various bug reproduction and root cause analysis systems.

Trigger Constraints: Ordering constraints give simple conditions regarding pairs of events to trigger a bug. These are easy to understand as we are within a constrained and well-defined event set $\mathbb{E}$. However, the trigger of events in $\mathbb{E}$ in practice is far from trivial. Many triggers occur "deep" in the system execution. In other words, the events in $\mathbb{E}$ only occur in the presence of specific node faults, link delays, state transitions, and other triggers. Trigger constraints allow us to characterize bugs in terms of the triggers required to bring them about. Intuitively, trigger constraints measure the "depth" of the bug - the more triggers the bug depends on and the lower the likelihood of each individual trigger, the more trigger-constraint the bug is.

Figure 1 shows a 2D plane with various bugs plotted on it according to the categorization we just described. The X axis represents the trigger complexity of the bug, and the Y axis represents the ordering constraint complexity. The labels on the Y axis are the subcategories of ordering constraints defined above. Unlike the Y axis, the breakdown on X axis is subjective and is based on number of necessary

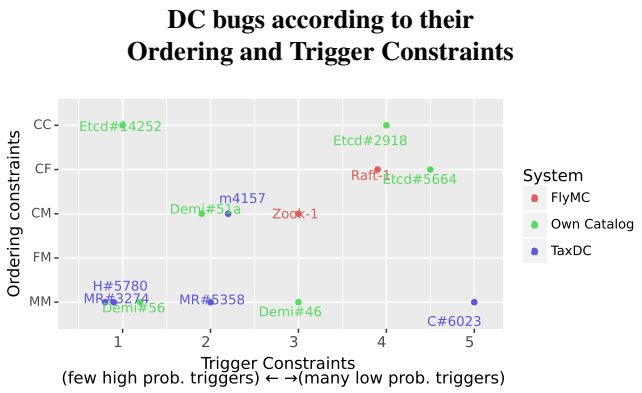

Figure 1: Various bugs on the 2D plane in terms of their Ordering and Trigger Constraints. Y axis represents ordering constraints, and the labels are the first initials of the subcategories Message, Compute and Fault.
X axis is the complexity of the trigger.
Though none of the bug catalogs we studied categorized bugs in this manner, we found this setup useful for reasoning about bugs and systems designed to prevent, detect or debug such bugs.

trigger conditions, likelyhood of their occurance under normal circumstances, and the difficulty to reproduce the bug in practice.

Below are short descriptions of some of the bug triggers:

(1) H5780: Timing of two high probability events at one node (*e.g.,* Message Send & Vote Receive)
(2) m5358: Timing of two high probability events across 2 nodes. Two Map-Reduce nodes must concurrently send to each other. Each receiver must receive the other's send after they have already sent
(3) DEMi 58: Upon transitioning from Candidate to Leader, the Candidate sends an ElectedAsLeader message to itself. It is possible that the newly elected Leader will receive ClientMessages before the ElectedAsLeader message is delivered, e.g. if it had the ClientMessages in its mailbox before the transition occurred. Upon receiving the ElectedAsLeader message, the Leader then reinitializes its nextIndex and matchIndex fields, overwriting their previous values that were more up to date [32].
(4) Etcd5664: Requires an etcd/raft node restart and an out-of-order receive message from the previous connection. The bug can also be triggered with an exceptionally slow sender and no restarts.
(5) c6023: Requires 3 message-message ordering constraints and several low-likelihood triggers.

4.2 TaxDC

TaxDC [17] is a large taxonomy of non-deterministic concurrency bugs collected from widely deployed open-source distributed systems (Cassandra, MapReduce, Zookeeper, and HBase).

Local concurrency bug catalogs existed before and were useful in understanding systems, finding strategies to avoid bugs, etc. TaxDC, however, was the fist comprehensive study of distributed concurrency bugs. Open-source distributed systems that were quickly gaining popularity at the time provided the authors of TaxDC with a good opportunity to carry out such a bug study.

TaxDC studies 4 systems from different categories defined earlier (Cassandra, MapReduce, Zookeeper, HBase). They start from the more general bug collection of the Cloud Bug Study [10] and distill it down to distributed concurrency bugs, which have been acknowledged as bugs by maintainers, are clearly described, and have been fixed.

The authors note that distributed systems bugs are at least as complex as their single-node counterparts. In the distributed setting, in addition to all local concurrency problems, we must deal with node failures, message delays, message ordering inversions, etc.

They then randomly pick a subset of 104 bugs which they further categorized according to various aspects.

The CBS study was conducted in 2011-2014, so TaxDC bugs also originated in this time frame. The authors report that similar distributed concurrency bugs were reported in the studied distributed systems after the study period. We can confirm through our own bug categorization efforts that at least until 2023 we have not stopped producing wild bugs in distributed systems. A lot has changed since 2014, but distributed system bugs seem equally hard to reproduce, understand, and fix.

4.3 FLyMC

The authors FlyMC [20] propose stateless/software model checking as a testing technique to uncover DC bugs in testing before deployment. However, the path explosion problem limits the scalability of current checkers, and they fail to scale under more complex distributed workloads. The authors introduce FlyMC, a distributed system software model checker that combines dynamic partial order reduction (DPOR) and bounded model checking (BMC) techniques to improve scalability and coverage.

Of the 22 bugs FlyMC uncovers or reproduces, 20 require ordering constraints to manifest. Of the 20, 9 require `msg-msg` ordering constraint to be reproduced, and 10 require `msg-fault`, `msg-compute` and `compute-fault` constraints in addition to `msg-msg` constraints. The remaining do not fit into this cataloging method. A few of the bugs found by the FlyMC model checker are plotted on Figure 1

4.4 Fixed It For You

Fixed It For You [27] proposes a method to automatically generate patches for distributed system protocol bugs. It requires implementing the system in a domain-specific language, but functions differently than a model checker. It can leverage correct and erroneous execution traces of real-world systems implementations to identify the bug's root cause. The authors use data provenance to aid in the bug-finding process. They introduce Nemo, a query language and framework for expressing debugging questions as queries over provenance graphs representing distributed program executions. When, in addition to provenance information and debugging questions, Nemo gets the protocol specification in its high-level specification language, it can go a step further and automatically propose fixes to the identified bugs. The authors provide a new taxonomy for 52 real-world distributed bugs from TaxDC.The authors evaluate Nemo at repairing errors of omission and identifying root causes of errors of commission on six protocol implementations. Among others, Nemo reproduced and proposed fixes for m3274, m4157, m5358 bugs from Figure 1.

4.5 Debugging Systems and Bug Constraints

Having positioned the bugs from studied catalogs on a map in Figure 1 we now highlight some aspects of relative positioning of the bugs on the map. We hypothesize that various automatic debugging approaches will be better at finding and reproducing bugs from a specific region on this map.

4.5.1 ***Logging:*** Since Dapper does logging at request level, it can be useful at gaining insights into `msg-msg` bugs. However, debugging deep bugs requiring multiple triggers across the nodes of a distributed systems may still be challenging, even when the bug only contains `msg-msg` constraints. This is because one has to be lucky that the particular messages involved in the bug have actually persisted (Dapper samples and logs a small percentage of requests).

Additionally, even assuming the necessary logs are persisted, it may be hard to tie together partitioned per-node logs with application-level semantics for debugging. This is particularly true when trying to debug bugs on the right side of the bug map (bugs with complex, multi-step triggers) as the bugs necessary to put together the complete story for a failure are likely scattered both in time and across system nodes.

4.5.2 ***Program Analysis Systems:*** ER, Gist and Pensieve seek to automatically find short and concise set of triggers for bugs in distributed systems. Since the systems try to isolate a small set of triggers for the failure, they face an exponential explosion of paths for failures that require multiple coordinated triggers. So, while approaches have a chance of surfacing bugs on the left side of the trigger axis of the bug map in Figure 1, they are unlikely to help debug bugs with multiple triggers.

Additionally, the approaches assume that the failure is visible immediately, so the system will know when to try to collect a sketch. This is not the case for many silent failures of the distributed system, such as a recent etcd corruption bug [2].

4.5.3 ***Formal Methods:*** In theory, given a precise system model and invariant specification of a system, a formal approach can detect the presense of a bug, regardless of its ordering constraints or trigger complexity. In practice, system models of pracical systems often make simplifying assumptions and model only a subset of messages in the system.

So, today's formal methods will likely help debug bugs on the lower left side of the bug map. The evidence from Nemo introduced in Fixed It For You is in line with this: Among others, Nemo reproduced and proposed fixes for the bugs m3274, m4157, m5358 from Figure 1.

4.5.4 ***Record/Replay Systems*** Since record-replay systems record and blindly reproduce machine-level events in a system, they are able to reproduce bugs with arbitrary trigger complexity with equal ease.

However, in order to be able to replay a specific bug, sufficient information must have been recorded about it. If recording has happened at `msg-msg` granularity, the replay cannot guarantee the replay of a `compute-compute` race.
Record-replay systems are immensely helpful for reproducing bugs, wherever they are applicable. Unfortunately, because of their high overhead, they often are not applicable in the setting of the distributed systems discussed in this work.

This highlights a particular dimension of the design space in record-replay systems. Various systems that record at different granularities and can replay different kinds of bugs, trading off recording performance with replay effectiveness. For example, RR and Castor are all the way up on the Y axis as those record `compute-compute` interactions (thread races). On the other hand, R2 [11] is somewhere in the middle, as it records at the application level.

The natural question then is "Is there a point on the trade-off above where record-replay systems record little enough to have acceptable overhead for production use, but still record enough to replay large classes of bugs"?

aiRR described in the next section is our answer to this question.

5 Design Overview

In this section, we give an overview of aiRR and highlight key design decisions. Our design decisions revolve around the following design goals:

(1) Maintain low runtime overhead so aiRR can run on production distributed systems

(2) Make the common case of integrating aiRR into distributed systems easy
(3) Make the general case of integrating aiRR into distributed systems possible

Past literature discussed above and our own findings detailed in Evaluation section indicate that existing record/replay approaches have prohibitive overhead when applied to distributed systems. Our first design goal is to improve on this in aiRR by making assumptions about the application under recording.

For the second design goal, we identify a common application development pattern and show how aiRR can be integrated into applications following the pattern. Most of the integration work in this case is within the bounds of the communication and IO libraries used by the application. This simplifies the common case of aiRR integration.

IO automaton: We notice that many practical distributed systems follow an implementation similar to the system model discussed in Section 2: These distributed algorithms are typically defined on a state machine in literature, and their real-world implementations often follow a similar structure. For the implementation, this means that the core algorithms are encapsulated in single-threaded blocks of code that receive some environment, run a computation, and return a result. Crucially, the core algorithms themselves do not have non-determinism. We leverage this insight to delineate what is necessary to record for successful replay, given the application semantics and the kind of bugs we would like to replay.

For example, to record the state transitions inside the raft state machine, it is enough to integrate aiRR so that it encapsulates the transitions of the raft state machine. This means that aiRR will be unable to replay and reproduce any bugs that are a result of nondeterminism outside of the core raft state machine. But, this also means that we will not pay the overhead of recording anything outside of the core state machine. If most of the events in this system bypasses the raft state machine (e.g. linearizable GET requests in etcd bypass it, as they do not require agreement among the raft members), then no recording will be necessary for the majority of the traffic volume.

In contrast, an application-oblivious record-replay system would be unable to differentiate between the two kinds of requests and would have to record all nondeterministic inputs, including the requests and inter-node traffic relating to linearizable GET requests in this scenario.

App integration via libraries: A key advantage of black-box record-replay systems is that they require no or minimal application changes so the cost of deployment and integration is quite low. Although an aiRR-like application-integrated approach improves on overheads, it comes at the cost of developer effort. With Design Goal (2) we aim to mitigate this as much as possible.

We notice that modern distributed systems applications have several structural commonalities, which we can exploit to make aiRR-integration easier for the developer. In particular, distributed systems 1) are **message-driven**, 2) use standard **RPC libraries** such as gRPC and CapnProto to encode/decode messages and 3) process each message in a **single thread of execution** within the application.

The message-driven architecture of these applications simplifies the process of adding instrumentation for recording purposes, as it reduces the number of necessary points for such integration. This approach also decreases the number of components that must be simulated during a replay. For instance, when replaying a conventional desktop text editing application, one needs to record all events from signal-controlled input/output devices and accurately reproduce them during the replay.

In contrast, in distributed systems, the main application flow doesn't rely on signals. Instead, these systems use a centralized logical queue that receives incoming messages. The application processes these messages sequentially, and then the results are sent to another queue. This means that the recorder only needs to know the order of incoming messages and the replayer needs to recreate those messages in the same order. This makes the integration task of aiRR simpler.

In addition to being message-driven, many distributed systems rely on RPC libraries to handle messaging (sending receiving, serializing, deserializing and ordering messages). As a result, the bulk of the modifications necessary to integrate aiRR can be done in these RPC libraries, outside of the application.

Note that modifying only the RPC libraries is not enough if the message order at the boundary of the RPC library is different from the message order at the serialization point of the application. So, some effort is still necessary to integrate an application with aiRR, even when it uses communication libraries already integrated with aiRR.

When applications are multi-threaded, there is nondeterminism that needs to be recorded even when all application inputs are deterministic as the non-deterministic thread scheduling can change program behavior during recording and replay.Past record-replay systems deal with this by either fixing the thread scheduling [19, 26] or recording all thread synchronization primitives such as locks, atomics, and assuming data-race-free applications [25].

Both approaches can result in significant degradation or result in inflated trace sizes when the application heavily uses synchronization primitives. The microbenchmark on Figure 2 discussed detail in Section 7 helps shows that there is a fixed cost to recording synchronization primitives that is the same order of magnitude as the primitive itself.

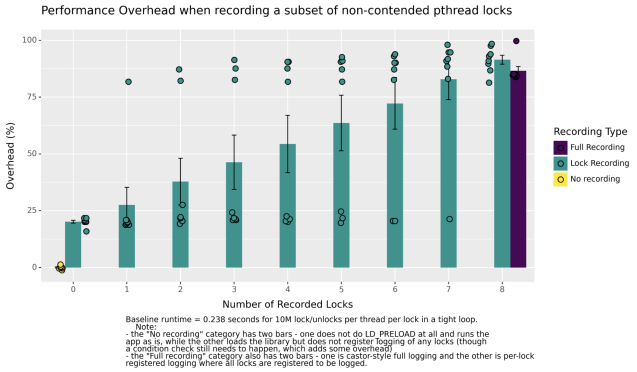

Figure 2: Lock recording overhead as a function of number of locks recorded

Informed by this, aiRR follows the approach of several prior works and provides an API for recording application-level synchronization. But, it in addition notices that the bulk of synchronization and multithreading is happening in the message-passing library, not in the core library. By recording the order of the messages after the messages have passed through the RPC library, aiRR can skip recording any RPC-library level synchronization.

When message processing inside the distributed system is happening in a single thread of execution, there is no scheduling related nondeterminism, so incoming message contents and order is the only nondeterminism necessary to be recorded. This helps reduce both storage and runtime overhead.

Large Buffer Overhead: When benchmarking SQLite's standard built-in benchmark, we noticed that while we can record all non-deterministic events with no noticible overhead, as soon as we started recording application-level buffers, benchmark ran 3 times longer. The problem is that non-deterministic events can be recorded in fixed 64 byte packets and amount to a small amount of copying per event. On the other hand, recording arbitrarily large application buffers requires a much larger copy and therefore has higher overhead.

Prior work [11, 25] suggests skipping the recording of these buffers by, e.g. skipping file IO operations altogether. This is often undesirable and may not solve the problem. It is undesirable as one may want to record only part of the buffer in the trace and not another part. A file-system API-level filtering will not allow such filtering. For example, an inter-node raft message may have some metadata about the current raft state and a set of Append logs. The application developer may want to record the metadata to make sure raft bugs can be successfully replayed but may be happy to record arbitrary message buffers in place of the actual Append entry content, when knowing the application logic does not depend on the content.

This may not solve the problem since not all IO comes through the file system. Many distributed systems communicate via messages, so large buffers may be in the form of incoming messages over the network. During replay these have to be reproduced, and unlike files on a file system, under no circumstances can be assumed to be there during replay without no extra work.

aiRR addresses this issue by allowing the application to selectively skip the recording of buffers at the application level. As the microbenchmarks in SQLite indicate, this can make the difference between allowing or prohibiting the system to run under aiRR recording in production.

The successful replay in aiRR is defined in terms of the recorded trace. A replay is successful if all recorded incoming messages were delivered to the application in recorded order, and aiRR observed the exact recorded responses from the application in the exact recorded order. If at any point during replay the application sends an unexpected message or expects a message not available on the trace, then the replay has diverged. This can happen with incomplete integration of aiRR into the system under recording.

6 Implementation

aiRR is packaged as a shared library that can be linked into the application. The same shared library is used during recording and during replay, so no recompilation is necessary. Whether the application runs in recording or replay mode is controlled via an environment variable. In the distributed system, each node of the system is recorded and replayed in isolation. During the recording, due to the application-level needs of the distributed system, all nodes need to be running at the same time and need to communicate. The recording at each node is oblivious to this application-level communication and simply records a separate independent trace for each node of the system. Note that aiRR can also be used for non-distributed system record-replay. Although many of our design decisions assume a distributed system setting and are designed to allow low overhead recording for production distributed system workloads. aiRR exports the low-level following API:

```
bool isReplay()
RecordReplay(const std::string &key,
             protobuf::Message& msg)
RegisterReproducer(const std::string &key,
                   ReproducerFunction f);
```

The function `isReplay` returns whether the system node is currently in recording mode or in replay mode. The function `RecordReplay` enables the application-level recording of external nondeterminism (environment, incoming messages, etc.).

During recording, it saves `key` and `msg` in the global trace. During replay, it expects to be called with the exact same key in the exact same order with respect to other calls to aiRR's recording API. If during replay the function is called out of order, it's call is blocked and delayed inside the aiRR shared library, until other calls that appear before the blocked one in the global trace arrive and are processed. `RecordReplay` is a kind of fence that records message ordering and enforces it during replay.

Note that `RecordReplay` assumes that the messages delivered during recording are also delivered during replay. This is fine for many sources of nondeterminism recorded via this API that initiate inside the node under recording (some examples are sending out response payloads, timer timeouts, and file IO-nondeterminism).

But it will fail for external messages that originated outside of the node under replay and will not be automatically delivered during replay since other nodes in the system are not necessarily running. `RegisterReproducer` API is used for these cases and tells the aiRR replayer how to mock-reproduce externally generated messages during replay. The API registers a callback for a unique key. Whenever ther replay process sees that the next event in the global key is one for which there is a registered hook, it calls the registered hook in the dedicated replayer thread. The definition of the reproducer function must itself interact with the aiRR API and call `RecordReplay` to consume the next message from the global trace.

Admittedly, the `RegisterReproducer` API is quite tricky to use - there are strict requirements for what kind of function the callback needs to be. Fortunately, this API is only necessary external RPC messages which pass through an RPC library and so it is enough to set up proper reproducers at RPC level, thereby freeing the core application developer from having to reason about external message reproducer callbacks.

Various libraries between the OS and the core of the application may use the aiRR API to record events that later will be replayed. All these events are serialized into a global order using their recording timestamps.

Note that aiRR also intercepts and `RecordReplay`'s certain libc functions out of the box. This simplifies the app-level integration of aiRR- e.g. since at the libc level we record and replay `getpid()`, there is no need to treat process PID as external environmental dependency and the application can assume that PID is fixed during recording and replay. We are working on expanding the set of APIs we can record at the libc level, thus simplifying the aiRR integration work necessary at the application level.

During replay, each node is typically replayed in isolation. The recorded trace has enough information to mock the presence of the rest of the distributed system for the isolated node under replay. The developer can still replay multiple nodes independently, attach debugger sessions to them, and debug some distributed invariant of the application. Note that coordination between nodes is not needed for the purposes of aiRR, during recording or during replay. Because the recorded trace is at a high level, it is possible to recompile the application in debug mode and replay that version of the application. The high-level nature of the recorded trace allows for debugging with the same trace but an updated binary. For example, the developer may wish to use a binary that has debug symbols for replay, instead of the original binary from production.

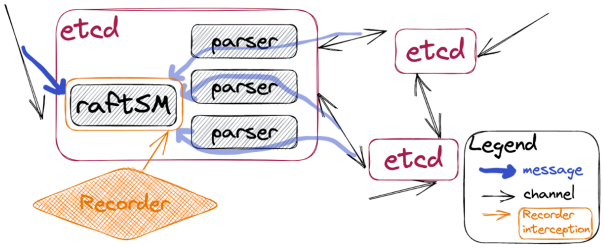

Figure 3

7 Evaluation

In this evaluation we try to answer the following questions:

(1) Can aiRR replay real world distributed system executions?
(2) Does aiRR reproduce bugs from real systems?
(3) Is aiRR's overhead low enough to allow recording in in production?

To answer these questions, we integrate aiRR into the popular etcd [8] raft-based distributed key-value store. Figure 3 shows what this integration looks like. An etcd cluster typically consists of 3, 5 or 2n+1 nodes. When run under recording, each node runs a version of the etcd binary that has aiRR integrated and enabled.

Each node of etcd receives and sends messages in parallel goroutines. These messages are parsed, serialized, and deserialized in gRPC and other application layers in parallel. These layers are represented as "parser"s on the diagram. The core logic for distributed state maintenance in etcd lives in the raft state machine that runs in a single thread. It accepts messages from the parallel parsers and posts outgoing messages to them. aiRR intercepts between this state machine and the parsers.

Avoiding recording locks: Due to the chosen location where aiRR is integrated, recoding does not at limit application level concurrency and does not have to record any coordination among the parallel parsers. The microbenchmark in Figure 2 shows that this can potentially reduce runtime overhead when the application uses extensive locking. The

microbenchmark measures the locking throughput overhead of recording variable number of pthread locks from a microbenchmark that uses these locks from 8 parallel threads. As the number of locks being recorded increases, we see that throughput overhead on threads using the recorded locks increases. When no locks are recorded but the recording infrastructure is built into the microbenchmark, there is a 20% runtime overhead. This is not fundamental and is the result of the chosen implementation technique. The relative overhead on lock instrumentation is lower in actual applications when the logging frequency is lower. The key takeaway of the microbenchmark is that recording coordination primitives can be expensive and should be avoided whenever possible. aiRR allows not recording coordination primitives in etcd parsers, while still enabling an exact replay of the recorded trace.

Q1: Etcd Integration and replaying production traces: aiRR integration into etcd required less than 100 lines of code changes inside the raft submodule of etcd and less than 20 lines of code changes outside of raft submodule in etcd. Most of the changes in the raft submodule of etcd amount to weaving aiRR between the message communication layer and the raft state machine. aiRR records total order and the necessary payloads of the messages at this boundary. Some changes are also necessary to record environmental inputs, such as random numbers and timers.

The few changes necessary outside of the raft submodule disable certain components of the node that try to actively run health-checks across nodes. Since these health-check messages are out of scope for recording, they will not be around during replay, and with these changes, we ensure smooth operation of the node during replay without the presence of these messages.

Since, by definition, aiRR is integrated into an application by its users, there are no system-level replay guarantees from aiRR. A wrong or incomplete integration can always result in divergence and replay failure.

The system can still be run under typical workloads and ensure that the recording and replay under aiRR is running properly. If an aiRR-integration can replay long traces of recorded distributed systems to completion, it can replay bugs contained in these tarces as well.

To gain some confidence that our integration of aiRR was correct, we built a test-harness on an etcd cluster to randomly change network conditions, restart nodes, add and remove participants, produce load etcd. We recorded each node of the etcd cluster and ensured that we can faithfully replay each node to completion.

Q2: overhead We measure the overhead of recording an etcd cluster while each node runs under aiRR. During the recording, we connect to the cluster through an external node and run a PUT workload at the capacity of the cluster with typical etcd key and value sizes [3]. We then compare the request throughput achieved with and without recording. Figure 4 shows that the throughput overhead of the recording is below 8% with a tail latency increase of <5%.

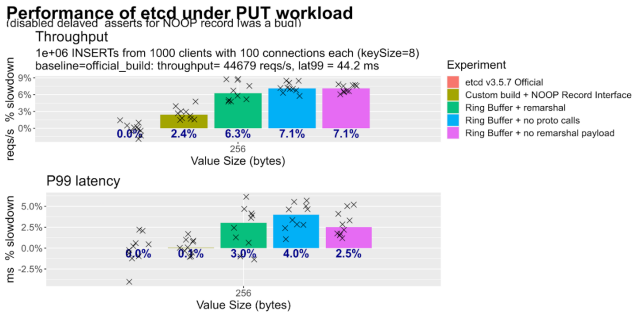

Figure 4

System	Issue Title
akka-raft #45	Candidate does not distinguish between votes from the same follower.
akka-raft #46	Candidate should check term in VoteCandidate messages
akka-raft #56	Nodes should not forget who they voted for
akka-raft #42	indexOnMajority is wrong
etcd #6744	raft: panic term should not be set when sending MsgReadIndex

Table 2: Highlighted bugs recorded and replayed using aiRR, from etcd and akka-raft distributed systems, from typical production-like cluster setups of these distributed systems

Q3: Bugs replayed. We reproduced a total of five bugs in our etcd setup, summarized in Table 2. Four of the bugs were taken from prior work on trace minimization [32] that found and reported these bugs in another consensus distributed system [1]. We ported these bugs to etcd and reproduced them under our test harness and ensured that the replay with aiRR recording succeeds.

We similarly reproduced one bug reported to etcd directly. When reproducing these bugs, the bottleneck was not aiRR but was our test harness - it was often not trivial to create the necessary conditions to trigger the bug. This step is required to obtain a aiRR trace and be able to test aiRR replay.

To reproduce these bugs, we set up a cluster of nodes running the distributed system with aiRR recording, using a system-specific typical production setting, recommended from the system documentation. We then manipulated various aspects of the cluster (link latency, packet drop, node availability, and crashes), until the bug we were trying to replay

was triggered. Depending on the bug under question, this took between minutes to hours. Whether we had a trace of only a few minutes or a multi-hour trace, we were able to replay them successfully using aiRR replayer.

7.1 Using aiRR for etcd: A Case Study

To give a sense of how aiRR is used in practice to reproduce and debug issues triggered in production, below is a description of how we debugged etcd#6744.

We use the test harness described above to introduce failures into the test clusters and record traces from all etcd nodes via our integrated aiRR system. We let the cluster run for a while, check standard application logs to confirm that something went wrong, and then collect all aiRR traces into a single node for replay and debugging. We also collect the application binary that was used in the faulty run. To gain confidence in correctness of aiRR integration into etcd, independent of the specific bug we are reproducing, we ensure that the replay of each node runs to completion without any divergence.

Now that we have the ability to deterministically replay in isolation any node from the cluster, we can use all of known debugging tools to analyze the deterministic replay and find the bug. In the following replay runs we attach a debugger to the process and step through the relevant stack frames. Very little is visible, since we are using the release build of the distributed system binary. Thankfully, aiRR traces are high level enough that we can replace the binary with the debug build and expect it to work. Doing so makes us navigate the source code as the replay is running under the debugger.

To allow for an even more powerful debugging experience, we run aiRR replay under RR recording. RR produces a much higher granularity recording by pinning the application to single CPU and recording all OS, environment, scheduling and instruction level nondeterminism. This almost triples the runtime of aiRR replay. But it allows time travel debugging in aiRR replay, powered by RR. Note that this kind of combination of aiRR with RR is possible because aiRR does not use the usual debugging mechanisms such as ptrace. Instead, aiRR is integrated into the application, and so the aiRR-replay can be recorded or monitored using the usual debugging frameworks.

With aiRR recording and RR-assisted aiRR replay, we found trigger of the bug (malformed message) with a handful of gdb commands. In particular, we set a breakpoint at the processing point of the first visible error message in aiRR replay, and run RR backwards-continue to understand how we got to that point.

8 Conclusion

We argue that distributed system bugs are here to stay, as existing work on debugging in various methods are not sufficient to detect them before production deployment or fix them quickly. We argue that even with state-of-the-art systems, understanding production failures of real-world production distributed systems—the likes of etcd, Dynamo [7], and Amazon S3—is still out of reach.

We propose an application-integrated approach to record production traces and enable an exact replay of the manifested bugs from production.

9 Bibliography

References

[1] 2015. akka-raft. (2015). https://github.com/ktoso/akka-raft

[2] 2022. v3.5 data inconsistency postmortem. (2022). https://github.com/etcd-io/etcd/blob/main/Documentation/postmortems/v3.5-data-inconsistency.md

[3] 2023. Etcd Performance Guide. (2023). https://etcd.io/docs/v3.4/op-guide/performance/

[4] Gautam Altekar and Ion Stoica. 2009. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM SOSP09, Big Sky Montana USA, 193–206. https://doi.org/10.1145/1629575.1629594

[5] Todd Arnold, Jia He, Weifan Jiang, Matt Calder, Italo Cunha, Vasileios Giotsas, and Ethan Katz-Bassett. 2020. Cloud Provider Connectivity in the Flat Internet. In *Proceedings of the ACM Internet Measurement Conference*. ACM, Virtual Event USA, 230–246. https://doi.org/10.1145/3419394.3423613

[6] Juan Benet. 2014. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561* (2014).

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[8] The etcd Authors. 2023. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. (2023). https://etcd.io/

[9] The Apache Software Foundation. 2023. Apache ZooKeeper: Centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. (2023). https://zookeeper.apache.org/

[10] 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. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Seattle WA USA, 1–14. https://doi.org/10.1145/2670979.2670986

[11] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. [n. d.]. R2: An Application-Level Kernel for Record and Replay. ([n. d.]).

[12] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding invariants of distributed systems: It'sa small (enough) world after all. In *Symposium on Networked Systems Design and Implementation-NSDI 2021*.

[13] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, Monterey California, 1–17. https://doi.org/10.1145/2815400.2815428

[14] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN '11)*. IEEE Computer Society, USA, 245–256. https://doi.org/10.1109/DSN.2011.5958223

[15] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure sketching: a technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, Monterey California, 344–360. https://doi.org/10.1145/2815400.2815412

[16] Jason R Koenig, Oded Padon, Neil Immerman, and Alex Aiken. 2020. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 703–717.

[17] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. 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*. ACM, Atlanta Georgia USA, 517–530. https://doi.org/10.1145/2872362.2872374

[18] Omar S Navarro Leija and Alan Jeffrey. 2019. Lightweight Record-and-Replay for Intermittent Tests Failures. *arXiv preprint arXiv:1909.03111* (2019).

[19] Christopher Lidbury and Alastair F. Donaldson. 2019. Sparse Record and Replay with Controlled Scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 576–593. https://doi.org/10.1145/3314221.3314635

[20] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, Dresden Germany, 1–16. https://doi.org/10.1145/3302424.3303986

[21] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. 2022. Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 787–803. https://www.usenix.org/conference/osdi22/presentation/luo

[22] Nancy A Lynch and Mark R Tuttle. 1987. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 137–151.

[23] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 370–384.

[24] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Transactions on Computer Systems* 35, 4 (Dec. 2018), 1–28. https://doi.org/10.1145/3208104

[25] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Xi'an China, 693–708. https://doi.org/10.1145/3037697.3037751

[26] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record And Replay For Deployability: Extended Technical Report. (May 2017). http://arxiv.org/abs/1705.05937 arXiv:1705.05937 [cs].

[27] Lennart Oldenburg, Xiangfeng Zhu, Kamala Ramasubramanian, and Peter Alvaro. [n. d.]. Fixed It For You: Protocol Repair Using Lineage Graphs. ([n. d.]).

[28] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[29] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017),

1–31.

[30] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 614–630.

[31] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. Rabia: Simplifying state-machine replication through randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 472–487.

[32] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. [n. d.]. Minimizing Faulty Executions of Distributed Systems. ([n. d.]), 20.

[33] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. [n. d.]. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. ([n. d.]), 14.

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

[35] H. Thane and H. Hansson. 2000. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*. 265–272. https://doi.org/10.1109/EMRTS.2000.854015 ISSN: 1068-3070.

[36] Wei Wang, Zhiyu Hao, and Lei Cui. 2022. ClusterRR: a record and replay framework for virtual machine cluster. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, Virtual Switzerland, 31–44. https://doi.org/10.1145/3516807.3516819

[37] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 357–368.

[38] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. {DuoAI}: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 485–501.

[39] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols.. In *OSDI*. 405–421.

[40] Cristian Zamfir, Gautam Altekar, and George Candea. 2011. Debug Determinism: The Sweet Spot for Replay-Based Debugging. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*. USENIX Association, Napa, CA. https://www.usenix.org/conference/hotosxiii/debug-determinism-sweet-spot-replay-based-debugging

[41] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, Shanghai China, 19–33. https://doi.org/10.1145/3132747.3132768

[42] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, Shanghai China, 565–581. https://doi.org/10.1145/3132747.3132778

[43] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution reconstruction: harnessing failure reoccurrences for failure reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 1155–1170. https://doi.org/10.1145/3453483.3454101