Applying a deterministic approach for distributed systems black-box testing

Edward Lee
Stanford University
edlee1@stanford.edu

Ivan Levchenko
Stanford University
vanya203@gmail.com

Abstract

Many existing distributed testing frameworks rely on randomized testing to identify issues. This makes reproducing such defects difficult since they are unable to replay the sequence of events that lead to the issue in the first place. And most frameworks that do allow for deterministic settings make many strong assumptions surrounding how the program is designed. We introduce a new testing framework that can be placed below any black-box distributed system implementation with very little modification to the implementation. It provides a deterministic, pattern-recognition-based approach to exploring the system's state space and is able to replay sequences of events that can lead to defects.

We implement a prototype of our framework on Linux, and evaluate the prototype on a self-implemented Raft implementation. The approach seems promising, though more work is required to evaluate the framework's effectiveness on third-party systems.

1 Introduction

Distributed systems have risen in popularity to meet demands for high availability and resilience to individual node failures, as well as network partitions [10]. However, the distributed nature of such systems can often lead to complex edge-cases and failure modes. Consensus protocols like Paxos and Raft are especially renowned for being difficult to understand and their implementations tricky to get right [6, 9].

Testing frameworks have thus sprung up in an attempt to test such complex systems. Some well-known frameworks currently used in industry include Jepsen [1] and Chaos Monkey [2]. While such systems can do a good job in identifying many consistency issues in existing distributed systems, there are a few caveats:

- It can be difficult to replicate potential defects identified due to the random and non-deterministic nature of the system [4].
- Such systems may require lengthy time investment into identifying appropriate tests and preparing these tests. In Jepsen's case, for example, many often enlist the help of its creator to construct and run such tests.
- Due to the randomized nature of such frameworks, it can be difficult to ascertain what states the framework has explored, and may take a lot of time to arrive at potentially interesting states.

We attempt to resolve such problems and more by constructing a testing framework that can be semi-transparently generate deterministic runs of a provided distributed system and use heuristics drive state exploration towards potentially new states. The framework does this through system call interposition — intercepting and modifying system call arguments and return values to carefully simulate a run-time environment with virtual clocks, injected node and disk failures, as well as message delays. The framework also attempts to intelligently explore new states using a simple heuristic for identifying similar, visited states and exponentially increasing the probability of failure should it seem like the framework is re-visiting previously checked states. This heuristic was developed in parallel with the framework, to speed up its development and allow independent verification of its effectiveness through simple use-cases like two-phase commit (2PC).

We implement a prototype of this framework on Linux, and test it on a self-written, bare-bones Raft implementation. We find numerous bugs in the Raft implementation during development of the framework, and re-introduce some of these bugs when evaluating the effectiveness of the framework. No third-party implementation is tested due to a lack of time.

The primary contributions of this paper are:

1. a semi-transparent testing framework of systems that can be easily configured to run on a local machine
2. deterministic replay of bug-causing traces through system call interposition
3. a heuristic for efficiently exploring the state-space

The remainder of the paper is organized as follows. Section 2 briefly discusses other work relating to checking distributed systems. Section 3 describes the framework developed to allow for deterministic runs. Section 4 presents the results of our independent experiments to identify an efficient heuristic for exploring distributed systems state-spaces. Section 5 presents the results evaluating on our custom Raft implementation. Section 6 discusses implications of our results and a conclusion.

2 Related Work

There are many frameworks for checking distributed systems, with varying levels of transparency and degrees of assumptions made regarding the system [7, 12, 13]. The work is most similar to MoDist [12]. MoDist is able to test unmodified distributed system code on Windows through similar ideas of system call interposition, a virtual clock, and failure injection. We attempt to extend on MoDist by investigating what it takes to port such ideas over to Linux, and use more simplified heuristics for efficient exploration of new states, rather than the complete traces used for DPOR by MoDist [3].

Techniques other than DPOR exist in the model checking space [5], and we take inspiration from these techniques when developing our simplified heuristic.

Other frameworks like FlyMC [7], Verdi [11], and Morpheus [13] make stronger assumptions about distributed systems by taking advantage of static analysis, re-writing systems in a formal language, and targeting specific languages, respectively. As such, while such frameworks likely provide efficient and deterministic model checking for the specific programs they target, they are not general solutions that can be placed under any distributed systems code.

3 Framework Overview

3.1 Architecture

Figure 1 illustrates the general architecture behind how the testing framework runs. A configuration file is provided to the deploying process with the commands required to start node and client programs. The deploying process spins up some number of orchestrators, and allocates each orchestrator some number of local loopback addresses to run the nodes on. The orchestrator controls all parts of the execution — from the scheduling of the nodes to when network messages get sent. Node scheduling can be done by stopping all nodes, and only resuming a single one at a time. And network messages can be managed by proxying all connections through the orchestrator, giving the orchestrator visibility into all messages being sent. Currently, for simplicity, each orchestrator manages a cluster of 3 nodes and 3 clients.

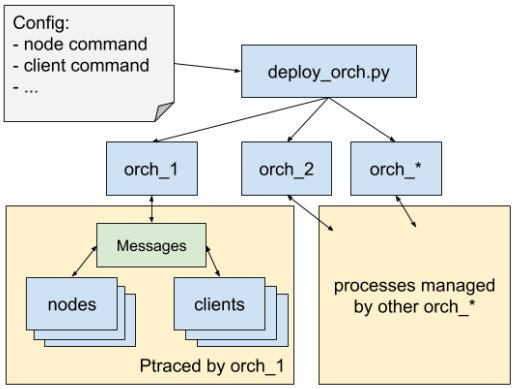

Figure 1: General architecture of the framework

3.1.1 System Call Interposition

The framework hinges on system call interposition, which we do through `ptrace`. The `ptrace` system allows a single process (the orchestrator) to receive signals whenever a process it is tracing (i.e. the nodes and clients) hits numerous events, including both when the tracee enters and returns from a syscall. This system allows for artificial failure injection, as when the orchestrator receives a signal that its tracee hit some traced event, it is able to modify everything from the syscall arguments and return value to the tracee's memory.

Not just that, `ptrace` is what enables us to achieve our goal of deterministic runs. Tracees will be stopped when they hit a traced event, which allows the orchestrator to run and process a single syscall at a time. It also means that the orchestrator can schedule or kill tracees as needed to allow for different interleavings and explore different parts of the state space.

Alternatives like eBPF and LD_PRELOAD were considered, but did not satisfy our requirements. Namely, eBPF does not allow for modifications of system call arguments and program memory, while LD_PRELOAD would likely fail on any system not using shared libraries like `libc`.

3.1.2 Virtual Clock

To ensure determinism of the system, and avoid any inadvertent timeouts due to the overhead our testing framework may bring to the system, we introduce a virtual clock by interposing on time-related syscalls like `clock_gettime()`. We do so by first disabling the vDSO (virtual dynamic shared object) for the given tracee by modifying the tracee's auxiliary vector. The vDSO in Linux allows time-related syscalls to run in user-space for efficiency, and thus would not be detected by `ptrace`. Once the vDSO has been disabled, we can intercept time-related syscalls just like the rest of the syscalls, and supply our own virtual clock-based time instead.

Primarily due to the lack of time, we make a restrictive assumption about how time is used in the tested system. Namely,

the system only times out on a polling syscall that fails to return any results. We would like to go further and handle implicit timers as mentioned in [12], but ran out of time to investigate this logic.

3.1.3 Network Interactions

As mentioned before, all network messages for a given cluster must go through the orchestrator. To do this deterministically, we take 2 measures.

First, we ensure all network-related requests are guaranteed to have been sent and received through waiting. When a node sends or connects to the orchestrator's proxy, the orchestrator waits indefinitely until the event is received to ensure that the send is not delayed and received at a later time. In the other direction, every time the orchestrator sends a message to a node, it waits an experimentally-derived length of 50 milliseconds to ensure it gets sent (as we have no way of ensuring the message gets received in its entirety by a non-blocking socket, for example).

Second, we track which sockets between nodes and the orchestrator have a connection. This allows us to deterministically inject failures and not rely on the OS, as Linux could allow a send to a failed node in one instance or fail it with `ECONNRESET` at the next. As far as we know, it is not clear what logic the OS uses to determine when to choose what. Instead, using this mapping between sockets, if we decide that a node should fail, we can ensure that orchestrator is notified of the node's failure and force failure on any connection or send attempts to the failed node.

3.1.4 File-System Failures

We want to check not just network-level errors, but storage-related errors as well. To do this, we need a way of modelling the file-system and identifying what exactly we want to test. As we didn't have much familiarity with the file-system space, we focus on a pretty basic idiom on how to atomically update a file, and model our file-system to be able to check it. According to LWN [8], it is recommended to follow the following steps to atomically update a file on most file-systems:

1. Create a temporary file in the same file-system
2. Write data to the temporary file
3. `fsync()` the temporary file
4. Rename the temporary file to the appropriate name
5. `fsync()` the containing directory

Since one `fsync()` is required to guarantee the contents are persisted, and the other to guarantee the metadata persisted, we attempt to model this separation in our checker.

To do this, we model the file-system as a series of checkpoints for individual files, as well as some list of pending rename operations. Each file checkpoint is determined by the contents of the file at `fsync()` time, with some special

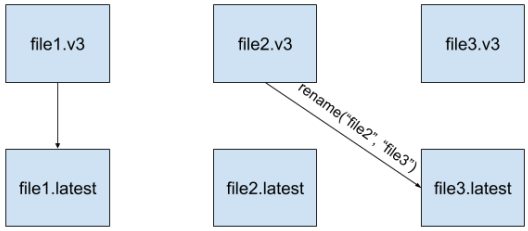

Figure 2: Example of what is tracked in our checkpointed file-system.

logic to handle renames. Each rename operation consists of a mapping from one checkpoint to another.

As illustrated in figure 2, on an `fsync()` of an individual file, our goal is to only update the contents of a file. As such, if there are any pending rename operations, the contents get written to the ancestor of the `fsync`'d file (so the contents of "file3" would get written to "file2.v3" in figure 2). Otherwise, a new checkpoint is created (e.g. if `fsync` was called for "file1" and "file2"). And on an `fsync()` of a directory, we want to update metadata, so any pending rename operations for the directory are executed in FIFO order.

This allows the checker to identify any deviations from the idiom since a missing `fsync` would cause either the contents or metadata to get out-of-sync.

3.2 Checking

While the framework is able to control individual node and client failures, as well as file-system failures, it does not have sufficient knowledge to validate that the distributed system is behaving as-expected.

Instead, the framework relies on client-supplied asserts that can be placed in three possible places: (1) within the node logic itself for node-specific invariants, (2) within clients for global invariants, and (3) in a validation command that gets run every 100 events. The validation command should not be necessary since the clients should be able to detect and assert any global invariants given sufficient logic, but validation can be helpful for early detection of errors.

3.3 State Space Exploration

We try out 2 different strategies for exploring the state space of inter-leavings between nodes and clients. The first is basic random exploration. Namely, we randomly make all decisions which include when to delay a message, when to kill a node, when to revive a node, and which node or client to run next. We use a standard Mersenne twister implementation for deterministic random number generation.

The second is based off of an exploration heuristic that is defined and further explained in section 4. The gist is that

Event	3-Count	Decision
1	n/a	d
2	n/a	d
3	{(1,2,3): 1}	d
1	{(2,3,1): 1}	d
2	{(3,1,2): 1}	d
3	{(1,2,3): 2}	d^2

Table 1: Simple example of how heuristic may be used.

we behave similar to the basic random exploration, but also track the historical behavior of nodes run and the syscalls they call. Based off of this historical data, we then attempt to identify and drive the system to new states by increasing the likelihood of failure of nodes if this bounded history has been seen before.

If these strategies aren't sufficient, there is an interface for clients to add additional logic for further methods to explore the state space. This interface is also how we implement deterministic replay, with each strategy recording a replayable trace of the decisions they make.

4 Exploration Heuristic

We attempt to reduce the exponential state-space of exploration of distributed systems by using an exploration heuristic to reduce what should be considered as a unique "state". We developed and evaluated our exploration heuristic in parallel with the actual development of the testing framework, and then integrated it in later. As such, we present the preliminary evaluation of the heuristic separately from the framework.

Our basic assumption is that the state of a system is cyclical, with similar states often re-appearing throughout a run. For example, while new leader elections in Raft [9] could be considered new states of the system, the actual state of the system is likely repeating states that have been seen before in previous leader elections. As such, rather than track the entire history of message sends to uniquely identify the state of a system as [12] might do, we instead track the number of occurrences of a sliding window of bounded traces. We can then use these counts to determine the course of action to drive a system to new states by exponentially increasing some "decision" parameter.

The key thing to note is that the heuristic thus requires 3 primary parameters — (1) the length of the sliding window, (2) identifying events, and (3) the decision being made. Table 1 provides a simplified example of how this might work. It has a sliding window length of 3, 3 different events, and some arbitrary decision parameterized towards driving the system to some new state.

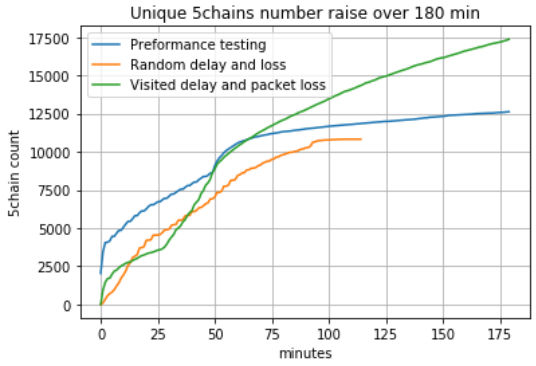

Figure 3: Number of unique 5-event traces using 3 different strategies

4.1 Evaluation

To evaluate this heuristic without the final testing framework on-hand, we instead use the heuristic in a simple delay-injection proxy system intended to non-deterministically inject delays into messages being sent across a network.

4.1.1 Evaluation on 2PC

We begin by evaluating this proxy system on a basic key-value store system built on top of two-phase commit (2PC).

We parameterize the heuristic with (1) a sliding window length of 5, (2) events determined by the tuple of the type of message (GET, PUT, VOTE, COMMIT, etc.) and its destination, and (3) the decision being made is the amount of time to delay the packet, or dropping if the delay is too high. Note that the source of the message is omitted from the event in order to reduce the size of the event space, and thus reduce both the memory required and the total number of states that we need to explore.

We compare the effectiveness of 3 different strategies in identifying new states over time. "Performance testing" runs the system as fast as possible, without intentionally adding delay into the network. This is often the only form of testing done in industry. "Random delay and loss" randomly introduces some static amount of delay to a message and has some probability of dropping a packet. This is reflective of the approach used by frameworks like Chaos Monkey [2]. "Visited delay and packet loss" is our heuristic as described above.

As can be seen in the results in figure 3, both performance testing and the random strategy are unable to find new traces and thus new states to explore over time. Instead, they plateau in the number of unique traces identified as time goes on, indicating that they are likely re-visiting already-visited traces. On the other hand, the visited heuristic is able to out-perform both strategies given enough time, and is able to consistently find new traces with no sign of plateauing even after 3 hours. We thus see that the visited heuristic seems capable of ex-

ploring novel states without requiring the parameter tuning present in the "Random delay and loss" strategy.

4.1.2 Evaluation on Raft LogCabin

Seeing that the visited heuristic seems promising, we attempt to use this delay-injection proxy to identify possible bugs in existing systems — namely an implementation of Raft called LogCabin written by the authors of the Raft paper [9]. The delay-based proxy system is much simpler to apply to new systems compared to the framework we implement due to the relaxation of the deterministic requirement. As such, we can use essentially the same exact configuration from the 2PC test for testing LogCabin.

We similarly use the same 3 strategies for evaluation, and are able to crash the system in 2 of the 3 strategies. Performance testing is unable to find any bugs or issues, which could be expected as the normal path is likely well-tested by this point. Both the random and visited strategies are able to crash the system by causing the nodes to hit their upper limit on number of open file descriptors. Whether this is a bug or not is left up to the reader, but it is worth noting that the visited strategy finds this defect much earlier than the random strategy. The visited strategy is able to find this bug after just 10 minutes, while the random strategy required 11 modifications to its delay parameter and a couple of hours [1].

Therefore, the visited heuristic and the strategies it enables look promising as a way to efficiently explore new states and possibly find new bugs.

4.2 Applying the heuristic to our framework

Due to the numerous differences between the delay-based system used for testing the exploration heuristic and the framework we create, we are unable to use the same parameters for the heuristic. For one, we want to be able to not just delay messages, but also fail nodes on specific system calls. Additionally, we have no information on the message format, and as such, cannot use message type as part of our events.

As such, our parameterization of the heuristic is as follows: (1) We maintain a window length of 5 to maintain memory- and time-efficiency. (2) Since we cannot distinguish messages, we instead focus on node behavior for our events. We thus consider the sequence of syscalls that a node performs while its running as a single "event". We assume there will not be too many of these since the behavior of nodes is likely quite rigid. (3) Due to lack of time, we were unable to add much control over messages. Instead, the decision being made is solely the probability of whether to fail a node or not.

[1] Video of finding the bug: https://drive.google.com/file/d/1So4m5AuBUFD0NLSZjOM9Wk_xeIVc6X-H/view?usp=sharing

Bug	Random	Heuristic
N/A	0	0
File-System	62	95
Stale Reads	91	80
Early Promotions	12	15

Table 2: Number of failures found for each bug and strategy used, out of 200 runs per experiments.

5 Results

We implement a prototype using both Python 3 and C++ [2], and test it only on a self-written Raft implementation [3] due to lack of time. There were too many syscalls and arguments to handle to generalize the framework to third-party distributed systems within the 10 weeks allocated.

5.1 Bugs found during development

While testing and developing the framework, we found a number of bugs in the Raft implementation. No counts are available, but the bugs primarily lay in the connection-management layer. One example is that the implementation made the assumption that only one TCP connection can exist between different nodes. This falls apart if nodes attempt to connect to each other at the same time. Such cases are likely to never appear in local testing, but are exposed due to the scheduling decisions being made by the testing framework.

The Raft implementation also did not follow the atomic-update idiom, which was caught. A protocol-level error was also caught, where a delayed VoteResponse message or AppendEntriesResponse message could cause errors in leader election and the log construction.

5.2 Finding injected bugs

In order to evaluate our framework on finding a diversity of bugs in distributed systems, we run 8 experiments of the framework using either a random or visited strategy on 4 different implementations: 3 with injected bugs and a control with no known bugs. The injected bugs are:

- a file-system bug where a file storing hard-state is modified directly rather than using the atomic-update idiom
- a protocol-level bug that enabled stale reads by allowing leaders to serve reads before committing an entry
- another protocol-level bug allowing pre-mature leader promotions by not checking term of a vote response

Each trial consists of 200 experiments run using the random strategy, and 200 more run using the visited strategy. All 3

[2] https://github.com/ed-w-lee/cs244b-testing
[3] https://github.com/ed-w-lee/raft-in-rust/

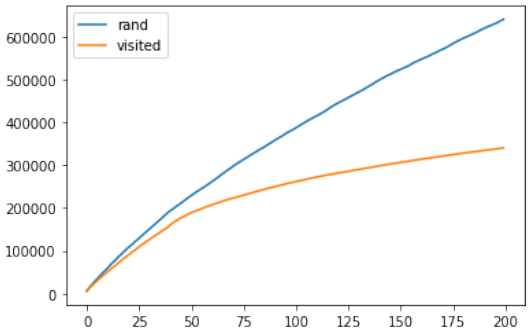

Figure 4: Number of unique 5-event traces in the control implementation over 200 experiments run on 40 orchestrators using our random and visited strategies.

of these bugs can be found regardless of the strategy used, as can be seen in table 2. From this, we can conclude that both strategies seem capable of identifying subtle bugs in both the file-system and protocol logic.

5.3 Effectiveness of Visited heuristic

Finally, we evaluate the effectiveness of our heuristic in finding and exploring new states. We do this by again running 200 experiments on an implementation with no injected defects. As can be seen in figure 4, contrary to our expectations, the heuristic-based approach was *not* able to out-perform the random approach in visiting new states. We are not entirely sure why this is, but we have a few suspicions. It may be that our heuristic definition was not sufficient in either its definition or its parameters to drive the system to new states. Or there may not have been enough experiments to out-perform the random strategy due to the increased state-space compared to our previously-tested 2PC example. Despite this, we still have high hopes in the potential of the heuristic for exploring new states, seeing as the heuristic did seem effective in non-framework testing.

6 Conclusion

Based on our results, we find that this deterministic framework seems promising in efficiently identifying defects early-on in the development of distributed systems. The independently-developed heuristic also seems promising for allowing the framework to steadily visit more and more states, though more investigation will be required to see why the final results with the framework do not reflect the preliminary results we received during the independent testing.

Unfortunately, despite the work done so far, we were unable to generalise the current prototype to work with any and all third-party implementations due to a lack of time. A few major hurdles we face in this effort include:

- Support for multi-threading: Most implementations use threading in some manner. We should be able to handle this by having `ptrace` trace clone events and trace new threads on spawn.
- Generalizing virtual clocks: Limiting assumptions are made for virtual clocks, and we would like to reach parity with MoDist [12] in being able to support implicit timers.
- More support for different I/O options: Linux has a wide array of options for I/O from `O_DIRECT` for direct access to files on a file-system, to `mmap`'ing files into memory for modification, to asynchronous operations.

Potential future work also includes trying out more complex exploration policies, and attempting to extract more information out of network messages to incorporate into the decisions made about when to delay or send a message.

References

[1] Aphyr. Jepsen.

[2] Michael Alan Chang and et al. Chaos monkey: Increasing sdn reliability through systematic network destruction. *SIGCOMM Comput. Commun. Rev.*, 45(4):371–372, August 2015.

[3] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices*, 40(1):110–121, 2005.

[4] Manish Rai Jain. How i solved jepsen with opencensus distributed tracing: A personal journey, 2020.

[5] A Kolchin, A Letychevskyy, and S Potiyenko. A static method for elimination of redundant dependencies in preconditions of transitions of formal models of transition systems.–2015, 2015.

[6] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[7] Jeffrey F. Lukman and et al. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Jeff Moyer. Ensuring data reaches disk.

[9] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[10] Alexandre Verbitski and et al. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[11] James R. Wilcox and et al. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Junfeng Yang and et al. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228. USENIX, April 2009.

[13] Xinhao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1141–1156, New York, NY, USA, 2020. Association for Computing Machinery.