# A Step-through Debugger for Distributed Systems

Douglas Woos

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

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Michael D. Ernst, Chair

Zachary Tatlock, Chair

Thomas E. Anderson

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

**Abstract**

A Step-through Debugger for Distributed Systems

Douglas Woos

Co-Chairs of the Supervisory Committee:
Professor Michael D. Ernst
Computer Science and Engineering

Assistant Professor Zachary Tatlock
Computer Science and Engineering

Designing and debugging distributed systems is notoriously difficult. For single-node systems, interactive debuggers enable stepping through an execution of the program and inspecting its state. For distributed systems, however, the execution control and state inspection facilities of traditional debuggers fall short. The execution of a distributed system is defined by the order in which events—messages and timeouts—are delivered; traditional debuggers do not allow developers to control this order. Additionally, significant system state resides on messages in transit rather than locally in program memory, and traditional debuggers are not able to display this state to developers. Existing step-through debuggers are therefore of limited utility to distributed systems developers.

The thesis of this dissertation is that a step-through debugger for distributed systems can bring the advantages of traditional single-node step-through debugging to distributed systems, helping developers to diagnose bugs and understand system behavior. We present Oddity: a graphical, interactive debugger for distributed systems. It brings the power of traditional step-through debugging—fine-grained control and observation of a program as it executes—to distributed systems. It also enables *exploratory testing*, in which an engineer examines and perturbs the behavior of a system in order to better un-

derstand it, perhaps without a specific bug in mind. A programmer can directly control message and failure interleaving. Oddity can be used on both executable system models and on system implementations. Oddity supports *time travel*, allowing a developer to explore multiple branching executions of a system within a single debugging session. Oddity includes a model checker for skipping tedious event sequences and for finding states matching particular predicates.

# TABLE OF CONTENTS

Page

# ACKNOWLEDGMENTS

I profoundly believe that when one closely examines grand accomplishments that seem to be the result of individual effort they are always actually the product of a whole community of people working together. My PhD, including this dissertation, is a perfect example of this phenomenon. I am tremendously thankful for the support and encouragement I've received from countless people. I recognize many of these people by name below, but am bound to have missed someone, for which I apologize.

First, I would like to thank my three advisors: Mike Ernst, Zach Tatlock, and Tom Anderson. It's a bit unusual to have three advisors, but I wouldn't have traded any one of them for the world. All three have taught me more than I can possibly describe about research, teaching, and life. Among other things, Mike taught me to look for conceptual contributions and to give concrete and actionable feedback; Zach taught me the very real value of endurance and confidence; Tom taught me pay attention to the bigger picture. I'd also like to thank the other members of my committee, Andy Ko and Jeff Heer, for their guidance.

I would like to thank my undergraduate thesis advisor, Tia Newhall. Tia introduced me to research and taught me the value of a good abstraction. I'd also like to thank some other mentors from my undergraduate days: Charlie Garrod, Rich Wicentowski, Doug Turnbull, and Benjamin Pierce.

I would like to thank some of the fantastic teachers I had before college, all of whom shaped my education: Tal Birdsey, Eric Warren, Bobby and Gerry Loney Dick Nessen, and Viveka Fox.

I am deeply thankful for the whole UW CSE community, especially the fine people

of the UW Programming Languages and Software Engineering Lab. James Wilcox and I started our academic careers together; I wouldn't be the researcher I am today without the time we spent flailing away in the Coq proof assistant trying (and sometimes succeeding!) to do the impossible. John Toman was the best office-mate I could ask for, a hypercompetent researcher, and a fantastic friend. I'd like to thank the guise, who know who they are.

I am proud to have been a member and, from 2018-2019, a steward of UAW 4121, the union for graduate students and other academic student employees at UW. I'm grateful to the union for looking out for my interests as a worker and for consistently standing up for what's right. I'd particularly like to thank David Parsons, Sam Sumpter, Leah Perlmutter, and Steven Pillen for teaching me how to organize.

Before I came UW I worked at GameChanger Media for two years. I'd like to thank my mentors at GameChanger: Kiril Savino, Ted Sullivan, Andrew Huling, and Tom Leach.

My parents, Jody and Dennis Woos, have been wonderfully supportive at every stage of my academic career. My mom taught me how to learn and my dad taught me how to program; both of them taught me how to be a person. My brother, Tim Woos, is the most impressive person I have ever met. Many thanks to all of them.

# Chapter 1

# INTRODUCTION

A distributed system consists of multiple physical machines, each of which communicates with the others by sending messages over a network. The Internet runs on various interacting distributed systems, from the Domain Name System used by all web clients and servers to distributed databases backing individual websites. With so many people depending on them for basic services as well as news and entertainment, it is vitally important that such systems be correct.

Developing correct distributed systems is difficult because of the combination of *asynchrony*—machines do not share a clock, making synchronization a challenge—and *failure*—machines can fail, and messages in the network can be dropped or arbitrarily delayed. A distributed system must maintain correctness in the face of arbitrary interleavings of message deliveries and failures. Bugs are likely to hide in the unusual failure cases. For example, the widely-used Raft consensus algorithm [49] was discovered to have a bug in its configuration-change protocol which would be triggered only by a specific interleaving of reconfiguration requests and failovers. Understanding distributed protocols even at a high level is notoriously challenging, and implementing large-scale systems that use such protocols is well beyond most developers.

Good developer tools can help developers discover, diagnose, and even entirely rule out classes of bugs, such as memory leaks, numerical errors, or out-of-bounds accesses. There is an additional class of bugs to which only distributed systems are vulnerable: *protocol bugs*, having to do with communication between multiple nodes (machines) in the system. Rather than leading to crashes, null pointer dereferences, or memory corruption, protocol bugs lead to violations of invariants the system should maintain. Figure 1.1

Figure 1.1: Examples of both protocol bugs and implementation bugs.

| **Protocol** | **Implementation** |
|---|---|
| A server will execute a non-idempotent operation twice if a client re-sends a request | The system leaks memory and eventually is killed by the operating system |
| A server allows a transaction to read data that were modified after the transaction started | The system's performance degrades over time because it stores crucial data in a linked list and depends on fast random accesses |
| A lock service allows multiple nodes to acquire the lock simultaneously if it detects that one of the nodes has failed, but does not correctly inform other nodes in the system of the failure | The system has a data race: multiple threads (running within the same node) access the same memory location, at least one of them with a write |

lists several examples of protocol bugs and implementation bugs.

Two particular properties make it difficult for developers to discover and diagnose protocol bugs. First, the state of a distributed system, rather than residing in particular variables at a particular machine, is distributed across multiple machines [50] and messages in the network [14]. Second, thanks to asynchrony, distributed systems are highly non-deterministic: messages and timeouts can be delivered in any order.

### *1.1 Step-through debuggers*

For single-node systems, developers have step-through debuggers (e.g., GDB or the Visual Studio Debugger). A step-through debugger (for the rest of this dissertation, "debugger" without qualification means "step-through debugger") helps a developer re-

produce and understand bugs by observing how the system's state evolves. Debuggers support several features:

1. Controlling the inputs (command-line arguments, keyboard input, etc.) to a program

2. Inspecting a program's state

3. Single-stepping through a program's execution

4. Execution pausing: pausing a program's execution when a particular condition is met

A developer invokes a debugger on a particular execution of a program, *controlling the inputs* to the program by providing command-line arguments and by interacting with the program normally (e.g., via standard input). Whenever the program is stopped, the developer can *inspect the program's state* (i.e., variables or memory locations). The developer can also alter this state—for instance, they can exercise alternative paths in a program by altering the value of an expression on which the program branches. They can *single-step* through the program's execution, executing one statement at a time. They can *pause a program's execution* via breakpoints, which cause the program to stop when it reaches a particular program point, or watchpoints, which cause the program to stop when its state matches a particular predicate.

One could imagine using such a debugger on a distributed system by attaching the debugger to every node individually, as in Figure 1.2. Unfortunately, this setup does not enable any of the four features needed to debug a distributed system. In particular, the network represents both a source of input and a location for program state, and is completely opaque to the developer. Input from the network determines each action the system takes. A debugger that does not have visibility and control over this input *cannot*

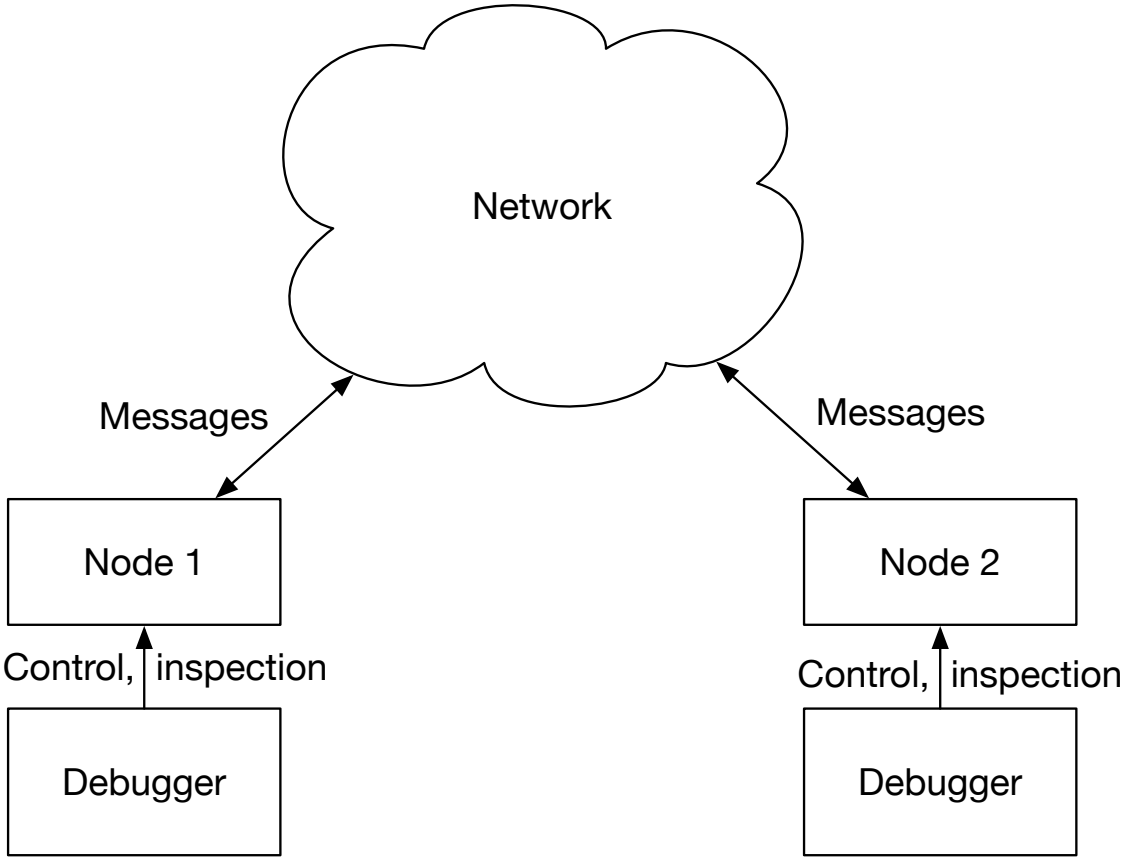


Figure 1.2: One way to use a traditional step-through debugger on a distributed system. Here, a developer has attached a debugger to each individual node, allowing them to control and inspect that node's execution. The nodes exchange messages over the network, over which the developer has no control.

enable users to inspect the full state of the system, single-step through an execution, or pause the execution of a program when particular conditions are met.

**Controlling input** The network's non-deterministic behavior—the ordering in which messages and timeouts are delivered—acts as an "input" to the system (similar to command-line arguments, files, or user input for a traditional single-node program). Inspecting,

controlling, and permuting this particular input is crucial to debugging a distributed system. Attaching a debugger to every node does not allow the developer to control this additional input.

**Inspecting program state** The developer can inspect each node's local state. However, there is no way to inspect messages *in the network*: messages that have already been sent but have not yet been delivered. This leaves the developer with an incomplete understanding of the state of the system.

**Single-stepping** The combination of network-resident state and non-determinism makes single-stepping through a distributed system's execution impossible using a traditional debugger. Consider a system being debugged as in Figure 1.2 where one node's execution is stopped on a system call that receives a message from the network (e.g., `recvfrom` on a UDP socket). If multiple other nodes in the system have sent messages to the receiving node, then this system call can result in any of those messages being received, or a timeout (if all of the messages are dropped or delayed, or if the developer is too slow). A traditional debugger does not provide control over which of these events happens.

**Execution pausing** Breakpoints and watchpoints enable a developer to instruct a traditional debugger to execute a program until particular conditions are met. Since the contents of the network are opaque to the debugger, developers cannot set breakpoints or watchpoints based on the network-resident component of the state.

### *1.2 Step-through debugging for distributed systems*

A step-through debugger for distributed systems should support the features described above: input control, state inspection, single-stepping, and execution pausing. In order to do this, such a debugger must be able to control the behavior of the network and

inspect the whole state of the system: not only each node in the system, but the messages exchanged between the various nodes.

This dissertation describes Oddity, a graphical step-through debugger for distributed systems. Oddity is designed to help developers diagnose and discover protocol bugs. Oddity is a *network-centric* debugger: it is designed specifically to debug and diagnose protocol bugs, leaving single-node bugs to traditional tools. Oddity provides all four of the crucial debugging features discussed above. First, Oddity allows the developer to control the behavior of the network by deciding which message or timeout should be delivered next. Second, Oddity displays the full distributed state of the system: state at each machine as well as the network messages and timeouts that are waiting to be delivered. Third, since developers can control the network's behavior, they can accurately single-step through the system's execution. Finally, Oddity implements execution pausing with a watchpoint implementation.

Some debuggers [10, 56, 31] for single-node programs feature time-travel: developers can backtrack over previously-executed statements. Oddity supports this as well, and—unlike previous work—also allows the developer to navigate a branching history of possible executions. This enables users to backtrack and make different choices about the order in which messages and timeouts are delivered, allowing the exploration and comparison of many different cases—for instance, all of the possible orderings of a few messages—without restarting the debugger. This is a crucial quality-of-life feature for debugging distributed systems, since understanding such a system necessitates observing its execution on multiple possible interleavings. Oddity implements time travel by replaying the sequence of events that lead to a state (and therefore assumes that it controls all of the non-determinism in the system it is debugging).

Most distributed systems are highly non-deterministic, with many messages and timeouts active at a given point in time. For such systems, it can be difficult or tedious to find a state where a given predicate holds. For instance, getting a new leader elected in an election-based consensus system such as Raft requires the delivery of a timeout fol-

lowed by a particular sequence of messages. For this use case, Oddity includes a *model checker* (see Chapter 7 for more on model checking), which finds executions satisfying user-specified predicates via exhaustive search. An explicit-state model checker such as Oddity's consists of two main parts: a mechanism for controlling the input to a system and a mechanism for checking predicates over the state of the system. Since Oddity provides control over the network input and inspection of the network's contents, and backtracking based on replay, Oddity's model checker is simply a driver for the debugger that implements exhaustive search over executions. Chapter 5 discusses Oddity's model checker in more detail.

In order to communicate with Oddity (in order to send debugging information and receive commands), systems use an event-based API. Depending on the complexity and architecture of the system being debugged, a developer can use this API in three ways. First, if the system is structured as an event loop that calls deterministic handlers for message and timeout events, the developer can simply replace the event loop with one that communicates with Oddity rather than the real network. Second, if the system's structure is more complex, the developer can create an *executable model*—a high-level version of a system that omits some implementation details—that directly uses the Oddity API. In previous work [45], such high-level models have been used to find protocol bugs and understand system behavior. Lastly, for some systems that use features such as blocking calls and background threads, a developer can use an adapter (included with Oddity) to automatically translate the system to Oddity's API. All three methods are described in more detail in Chapter 4.

### *1.3 Hypothesis and contributions of this dissertation*

The thesis of this dissertation is that a step-through debugger for distributed systems can bring the advantages of traditional single-node step-through debugging to distributed systems, helping developers to diagnose bugs and understand system behavior. In order to support this thesis, we make the following contributions.

**The Oddity debugger and user interface** Oddity's core feature is a debugger supporting state inspection of the entire distributed system state (nodes and messages) and allowing developers to control all inputs to the system, including the ordering of message and timeout delivery. Oddity presents these features via a novel graphical user interface. Oddity defines an API for distributed systems to interact with the debugger interface.

**The Oddity model checker** Given a predicate, Oddity's model checker will attempt to find a sequence of events leading to a state where that predicate holds. Unlike previous model checkers, Oddity's enables *interactive* use: a developer can search for a state using the model checker and then explore manually or re-invoke search for subsequent states. An integrated model checker is necessary for a good distributed systems debugging experience, but it may be useful for traditional single-node debuggers to integrate a model checker as well in order to enable such exploration.

**Debugging system implementations with Oddity** Oddity can be used to debug existing distributed system implementations. Oddity controls these implementations via a new system-call interposition system, MajorTom. Because Oddity is a network-centric debugger, MajorTom instruments only those calls necessary to present the user with an accurate, high-level view of the messages being exchanged in a system. We have implemented support for a subset of the Linux system call interface, which enables developers to debug realistic systems at the protocol level.

**Evaluation of step-through debugging for distributed systems** Students have used Oddity in three distributed systems courses (two offerings of an undergraduate-level course and one offering of a master's-level course). Our qualitative and quantitative observations provide evidence that step-through debugging in general, and Oddity in particular, can be useful for finding and diagnosing bugs in distributed systems.

## *1.4 Outline*

The remainder of this document is structured as follows. Chapter 2 demonstrates Oddity via a running example. Chapter 3 presents Oddity's core features, implementation, and API. Chapter 4 presents several ways for developers to use the Oddity API to debug systems. Chapter 5 presents Oddity's integrated model checker. Chapter 6 discusses quantitative and qualitative evaluations of Oddity. Chapter 7 discusses related work. Finally, Chapter 8 concludes with a discussion of Oddity's current limitations and potential avenues for future work.

# Chapter 2

# AN EXAMPLE ODDITY DEBUGGING SESSION

This chapter introduces Oddity's core ideas and interface via a running example: debugging an implementation of Raft [49]. Raft is a consensus protocol, which is a key component in the construction of strongly-consistent distributed services. A consensus protocol enables a cluster of nodes to agree on a sequence of values, despite node failures and arbitrary message delays. To support changes in the nodes participating in the state machine consensus, Raft includes a reconfiguration protocol in which both the new and old sets of nodes must agree on any new configuration. The reconfiguration protocol can be triggered manually by a system administrator or automatically by a cluster management system. Raft is widely deployed in industry.

Ongaro's dissertation [46] includes a simplified reconfiguration protocol designed for single node configuration changes (i.e., a single node joining or leaving a Raft cluster). Several years after publication, researchers discovered [47] a bug in this simplified protocol: in a cluster with an even number of members, if two competing reconfiguration requests occur with a leader election in between, the cluster can lose data. A simple fix, proposed when Ongaro publicly announced the bug [47], is to require that new leaders commit an entry to the log in the old configuration before committing a new configuration. Several months passed between the bug being identified and the fix being announced.

For explanatory purposes, we imagine a Raft maintainer has been informed of the existence of the buggy execution; using Oddity, they are trying to determine why it happens and how it can be fixed.

Figure 2.1 shows an execution leading to the Raft bug. First, a $S_1$'s election timeout

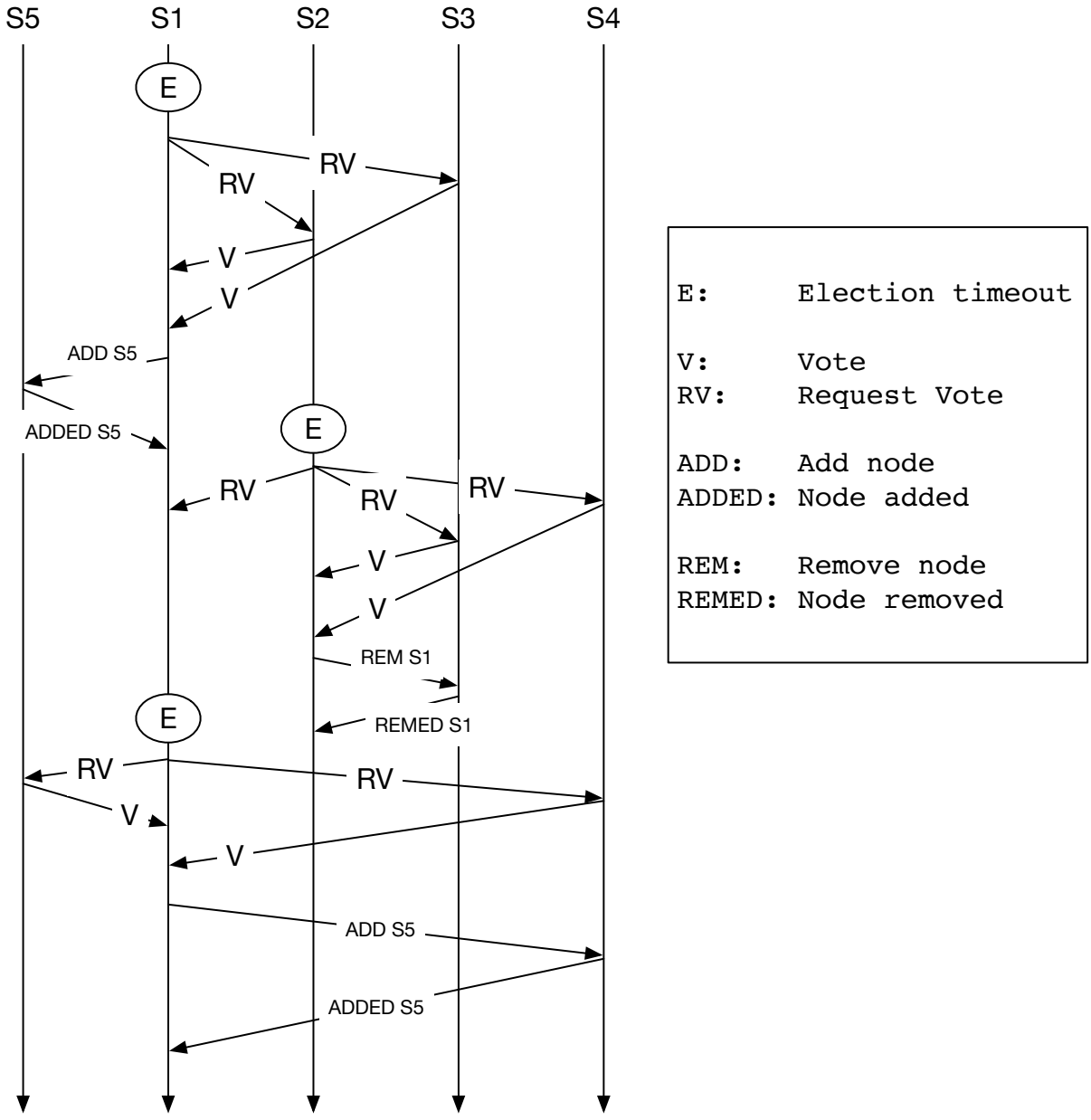


Figure 2.1: A space-time diagram illustrating a Raft execution leading to the reconfiguration bug. Each vertical line represents in a node in the system, and arrows between them represent messages. The circles represent messages received from clients of the system (elided for presentation).

is fired, causing it to attempt to become the leader by sending vote requests; having received enough "Yes" votes in response, it is elected leader in a 4-node cluster by a majority including itself, $S_2$, and $S_3$. Then, $S_1$ starts to replicate a new configuration that adds a fifth node, $S_5$. In the buggy single-server reconfiguration protocol, each server uses whichever configuration is latest in its log (regardless of whether it is committed). The leader sends this new configuration to $S_5$ (shown on the left of Figure 2.1) as well as the rest of the cluster (assumed to be delayed or dropped in Figure 2.1). After this configuration is replicated to $S_5$, $S_2$ starts an election and is elected with votes from $S_3$ and $S_4$. This might occur, for example, if the reconfiguration messages from $S_1$ are delayed to those nodes, e.g., due to a temporary network outage. (Consensus should work even when nodes incorrectly judge that other nodes have failed.) Now that $S_2$ is leader, it starts to replicate a new configuration that removes $S_1$ from the cluster, leaving the three nodes $S_2$, $S_3$, and $S_4$ (since the configuration with $S_5$ was never replicated to $S_2$). It successfully replicates this configuration to $S_3$, at which point it can commit the configuration since it is on a majority of nodes in the new cluster of three nodes. Now $S_1$ starts another election, and becomes leader with votes from $S_4$ and $S_5$. It can now finish replicating its configuration adding $S_5$ to the whole cluster, which overwrites $S_2$'s committed configuration. This is a violation of a crucial Raft safety property: once an entry is committed, it should never be overwritten.

Without Oddity, the Raft engineer has several options to reproduce and diagnose this failure. They could examine the code and try to imagine an execution that would trigger the bug, but this is both time-consuming and error-prone. They could design an automated test to find the issue, but testing distributed systems is notoriously difficult [40]. Since the issue depends on a failover, the test environment would need to simulate a temporary network partition. The test environment would also need to ensure that messages are delivered in a specific order with respect to other messages and the network outage. The engineer would also need to write an oracle that determines whether the bug has in fact been triggered (i.e., whether data are lost). Finally, the engineer could run

their code in a traditional debugger and attempt to trigger the issue. Doing so, however, would still require simulation of failures and control over the order in which messages are delivered.

The rest of this section shows how Oddity makes the Raft engineer's task easier, illustrating Oddity's functionality via an example debugging session: reproducing and diagnosing the raft reconfiguration bug. A screen-cast version of the debugging session can be found at `http://oddity.uwplse.org`.

### *2.1 System setup*

Oddity assumes that the system being debugged is implemented as a set of event handlers: deterministic functions that can read and write the node's state, send messages, and set timeouts through the Oddity API (detailed in Chapter 3). Implementing the Oddity API in a distributed system is discussed in Chapter 4. The system includes five Raft nodes, four of which are aware of each other's existence and are in a cluster. It also includes a client node that, in response to timeouts, sends reconfiguration commands to the Raft cluster.

### *2.2 Finding a buggy execution*

When the engineer starts Oddity on their system, they will see a screen similar to Figure 2.2. Each node has an "inbox" next to it, which contains both messages sent by other nodes and also timeouts the node has set itself. At the beginning of time, no messages have been sent, so each node's inbox contains only election timeouts waiting at that node (including $S_5$, which has not yet been added to the cluster). When the system is running normally (i.e., not being debugged in Oddity), these timeouts are fired when a node has not received a message from a leader for sufficient time. Using Oddity, the engineer can deliver the election timeouts at any time.

The engineer will first need to get $S_1$ elected leader. They can click on the `E` (election)

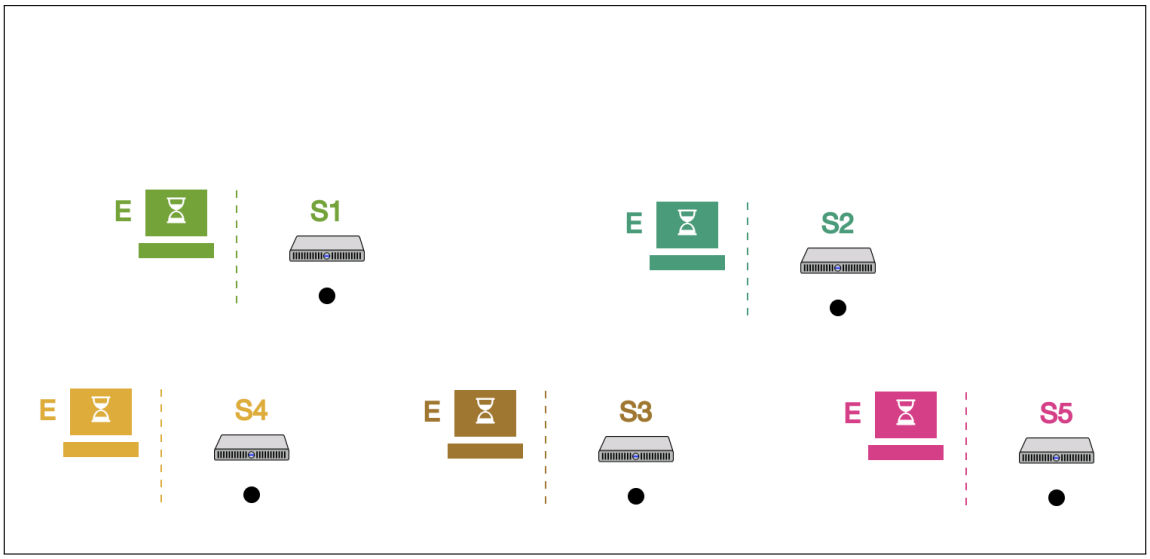


Figure 2.2: The initial state of the Raft system in Oddity. Each node has a timeout in its inbox.

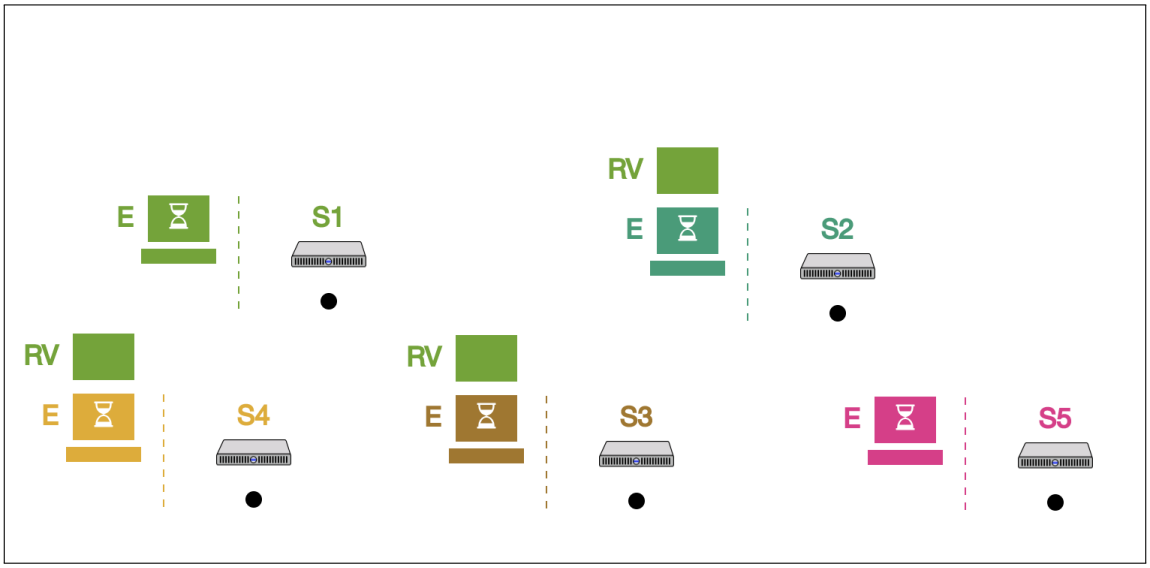


Figure 2.3: The state of the Raft system after $S_1$ starts an election. $S_2$, $S_3$, and $S_4$ have `RV` messages in their inboxes. The messages have the same color as their sender ($S_1$).

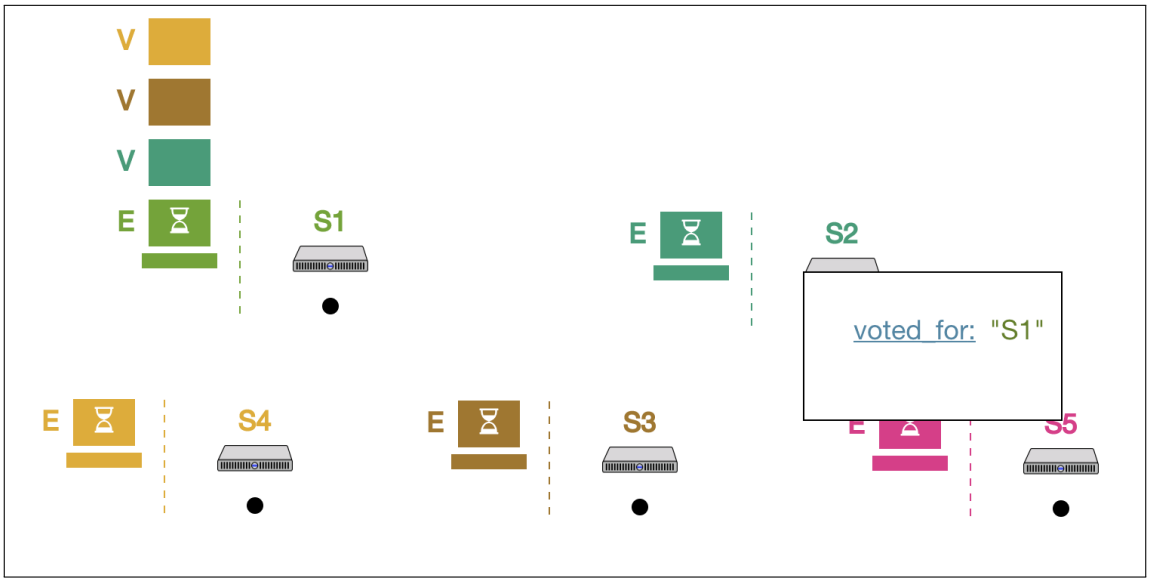


Figure 2.4: The state of the Raft system after $S_2$, $S_3$, and $S_4$ respond to $S_1$'s vote request. The votes from those nodes are in $S_1$'s inbox, with colors corresponding to the sending node. The engineer has clicked on $S_2$ to expand its state.

timeout in $S_1$'s inbox to deliver it, causing $S_1$ to send `RV` (Request Vote) messages to the other nodes in the initial configuration (excluding $S_5$, which has not yet been added). The resulting state of the system, with a `RV` message in each node's inbox, is shown in Figure 2.3. These messages are now waiting to be delivered.

The engineer can click on each `RV` message to deliver it, causing the receiving node to respond to S1 with their `V` (Vote) messages. In Figure 2.4, these messages have been sent and $S_2$'s state is expanded, showing that it voted for $S_1$. The engineer can click on these vote messages to deliver them to $S_1$. Since Raft requires a quorum (in this case, three nodes) to elect a leader and $S_1$ has already voted for itself, once two of these votes are delivered $S_1$ considers itself elected.

Now that $S_1$ is the leader, the engineer can investigate the reconfiguration bug. They can eliver a timeout to the client (not shown), causing it to send a reconfiguration request

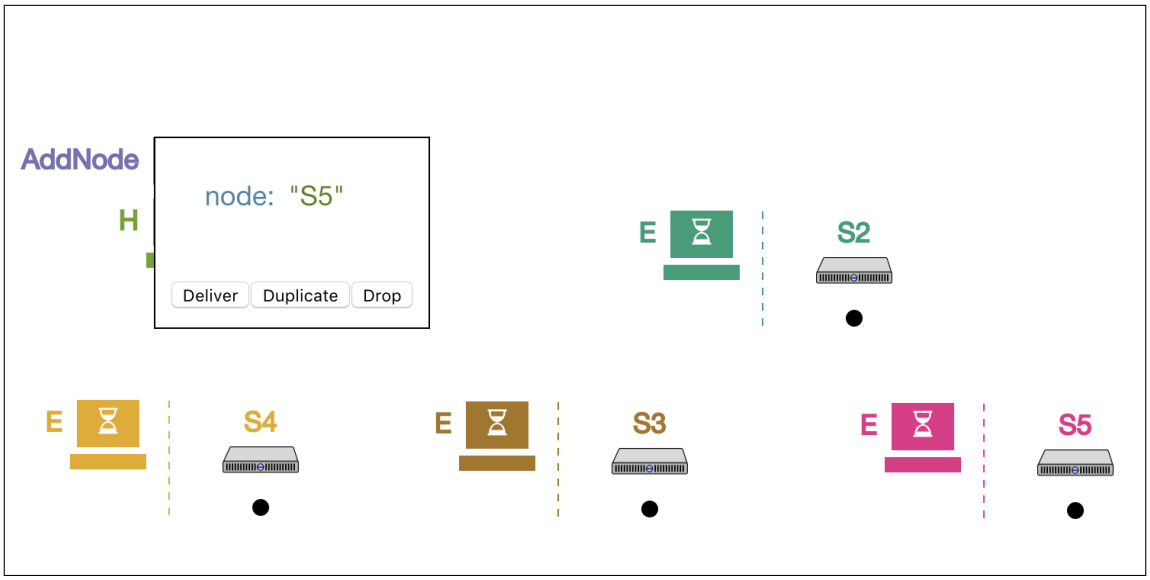


Figure 2.5: The state of the Raft system after the client sends its reconfiguration request. The request is open for inspection. As shown, the engineer can choose to duplicate it or drop it rather than delivering it.

to add $S_5$. They can inspect the request by clicking on it, as shown in Figure 2.5. Once the request is delivered, the leader will try to commit this new configuration to a majority of the new configuration per the single-node reconfiguration protocol.

Once the new configuration has been replicated to $S_5$, the engineer needs to trigger a new leader election in order to continue following the counterexample. They can do so by delivering the `E` timeout to $S_2$, triggering an election.

The rest of the leader election is elided for brevity. The engineer now delivers another timeout to the client, causing it to send a second reconfiguration request: to remove $S_1$ from the cluster. The reconfiguration request is delivered at $S_2$ and $S_2$ replicates the new configuration to $S_3$; this configuration is now committed, because it has been replicated to a majority of the new cluster.

The counterexample now calls for $S_1$ to start a new election. The engineer triggers

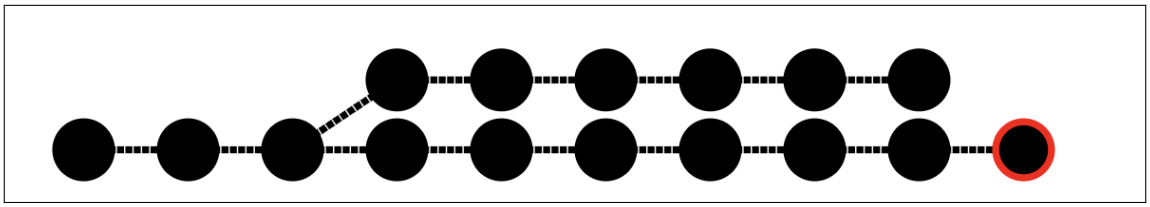

Figure 2.6: Oddity's history view. Each previously explored state is shown as a dot. If multiple transitions out of a state are explored, Oddity displays a branch.

this by delivering $S_2$'s `RV` message (ensuring $S_1$'s term is up to date) and then the `E` (election) timeout. After getting elected, $S_1$ replicates the configuration with $S_5$ to the rest of the cluster. Crucially, $S_1$ replicates the updated configuration to $S_2$, overwriting a previously-committed entry and demonstrating that the Raft implementation is buggy.

### *2.3 Backtracking*

The reconfiguration bug can be fixed by requiring that a leader replicate an entry in its term before attempting to reconfigure the system. The engineer could test this potential bug fix without changing the Raft implementation by exploring an execution in which $S_2$ attempts to replicate a no-op entry in its old configuration before it receives the request to reconfigure the system.

Oddity allows the engineer to explore this alternative execution without restarting the debugging session. The engineer can click on any previous state in the history view shown in Figure 2.6 in order to reset the system to that state. They can then explore other executions starting from that state. Using Oddity's execution history view, the engineer can go back to the point just before $S_2$ started to replicate the command removing $S_1$ and instead deliver a heartbeat timeout to $S_2$, causing it to attempt to replicate a no-op entry in the old configuration. In order to proceed, $S_2$ must replicate the no-op entry to at least three nodes (e.g., $S_2$, $S_3$, and $S_4$) before it can attempt to remove $S_1$ from the replica set. At that point the pending reconfiguration with $S_5$ will not succeed, since

$S_1$ will not be able to be elected leader until its log is up to date with the rest of the cluster. The engineer now has some evidence that the proposed bug fix (requiring that a leader replicates an entry in its term before attempting to reconfigure the cluster) works: it prevents this particular faulty execution.

## *2.4 Model checking*

As discussed in Chapter 5, Oddity includes a model checker in order to enable exhaustive search over possible execution traces. Developers can instruct Oddity to find a sequence of events, starting in the current state, that leads to a state where some desired predicate holds. In the Raft example, a developer could use this feature to elect the initial leader, $S_1$, by specifying the predicate `S1.state = ``Leader''` and running the model checker. This avoids the tedium of having to choose individual messages to deliver. They could do the same thing for the two subsequent leader elections.

## *2.5 Real systems and executable models*

The debugging session described in this chapter could be on either an executable model of a user's Raft-based system or on the system itself; the user's debugging experience would be the same in either case. Chapter 6 describes how Oddity can debug LogCabin, the reference implementation of Raft. LogCabin uses a different reconfiguration protocol (the multiple-node protocol described in the Raft paper rather than the single-node protocol described in Ongaro's thesis); it does not suffer from the bug discussed in this chapter.

# Chapter 3

# ODDITY DESIGN AND IMPLEMENTATION

Oddity's design is informed by the goals discussed in the previous sections. Oddity must enable developers to control and inspect the behavior of the network in order to debug their systems' behavior on various message and timeout orderings. Rather than relying on a real network (and real time) for message and timeout delivery, Oddity runs systems in a virtual network and then gives users control over the virtual network's behavior. This chapter discusses the design and implementation of Oddity: Oddity's high-level architecture, its novel graphical interface, and its implementation.

Figure 3.1 shows the architecture of the Oddity debugger. Oddity consists of two cooperating components: a browser-based frontend that displays the user interface and a backend, split between the browser and the server, that tracks the system's state, implements time travel, and communicates with nodes in the system. Oddity's graphical interface is independent of its backend; either could be replaced without changing the other.

System nodes communicate with Oddity via the Oddity API, an event-based interface Oddity uses to control systems and obtain debugging information. Chapter 4 discusses several mechanisms nodes can use to implement this API. Oddity is intended to be used during development, so in general all nodes will be running on the same machine; this is not, however, a hard requirement as each node communicates with Oddity independently.

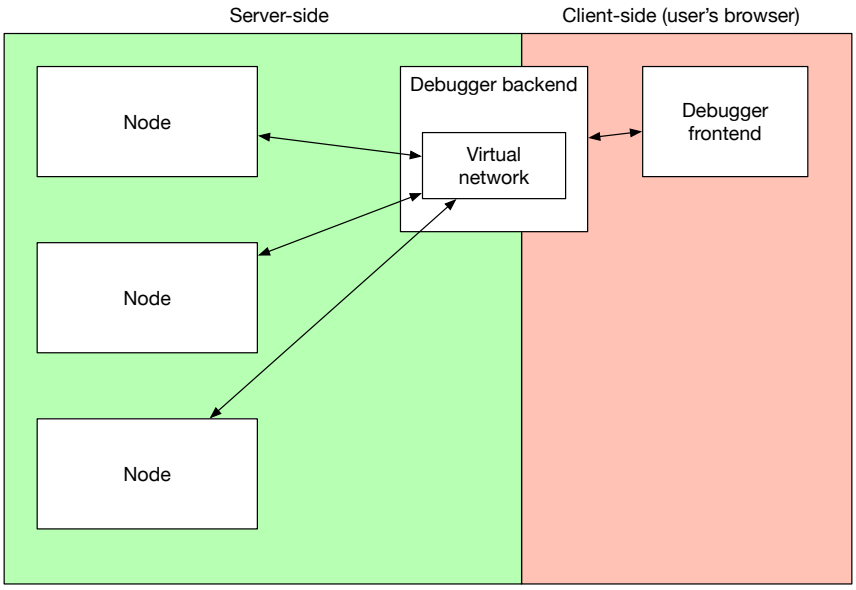


Figure 3.1: The architecture of the debugger implementation. The debugger backend implements the virtual network and communicates with individual nodes using the Oddity API. The debugger frontend, running in the browser, communicates with the backend. Most of the logic runs in the browser, allowing the backend to serve as a thin communication layer.

## *3.1 Oddity backend*

The debugger backend implements the virtual network, which contains the network-resident state of the system (in-flight messages and timeouts). It also records the event history. An event is a message delivery, a timeout delivery, or the special "start" event representing the beginning of time. When the user tells the frontend to deliver a message or a timeout, the backend records this event and then sends the message or timeout to the appropriate node. When the backend receives the response, it tells the frontend to update the display to reflect the new messages and timeouts and the modified local state.

When the user navigates to a previous state in the history display, the debugger back-

end resets the system to that point by replaying all of the events that led to that state (including the special "start" event). The user can then explore alternative executions starting from that state. Backtracking via replay is possible under the assumption that Oddity controls *all* of the inputs to the system, including any non-determinism. If this is not the case (e.g., if the system uses randomness), replay will not work correctly. Chapter 4 discusses ways of meeting this constraint even with systems that use randomness internally.

System nodes communicate with Oddity via the API shown in Figure 3.3. As discussed above, Oddity assumes that the system will respond deterministically to a given sequence of events. Oddity also assumes that after responding, a node blocks waiting for the next Oddity message and does not continue processing. Chapter 4 discusses several ways of implementing the Oddity API in a distributed system.

## 3.2 *Oddity graphical interface*

Oddity's graphical interface, shown in Figure 3.4, is designed to enable engineers to easily explore executions of distributed systems, including failure cases. As discussed in Chapter 7, many distributed systems visualizations are based on space-time diagrams (see Chapter 7). Others are system-specific, visualizing some particular structure in the state (including messages and nodes) of a system. Still others are system animations, where messages fly between nodes in an abstract representation of a network. The use-case for Oddity's interface—controlling and inspecting the execution of a system—requires a novel visualization:

1. Unlike visualizations based on space-time diagrams, it should enable *detailed inspection of a single global state of the system* (including the contents of all messages and the local state at every node).

2. Unlike system-specific visualizations, it should be *application-agnostic*. A user should

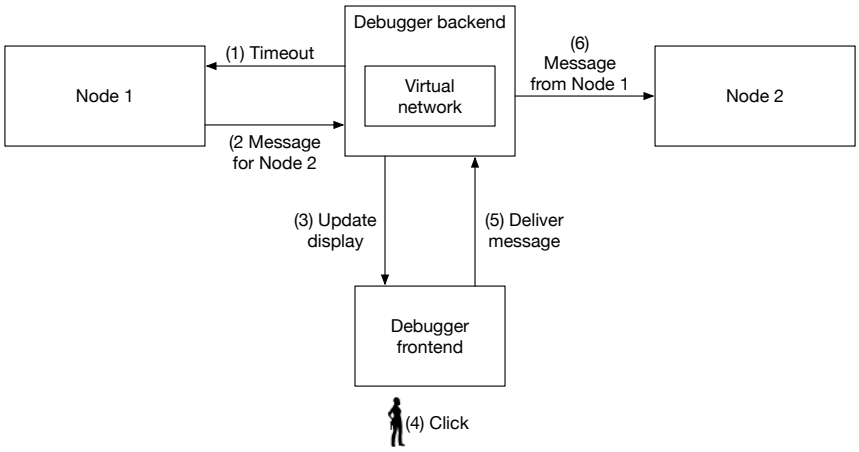


Figure 3.2: An example of Oddity components communicating to deliver a message. When Node 1 receives a timeout, it produces a message for Node 2. Node 1 replies to the debugger backend with this message. The debugger backend the frontend to display it in Node 2's inbox. Control is returned to the user. When the user clicks the message (which they could do immediately or after delivering other messages and timeouts) the frontend notifies the backend, which sends the message to Node 2.

be able to graphically debug their system without developing a system-specific visualization.

3. Unlike animation-based visualizations in which messages travel between nodes, it should *highlight asynchrony*. Users must be able to arbitrarily delay and reorder messages and deliver timeouts even if no failures occur.

Oddity's frontend addresses each of these requirements.

**Single-state inspection** Oddity's graphical interface is geared towards representing a single state of the entire system—including node states, in-flight messages and potential timeouts—in detail, while also enabling users to navigate a branching execution history. Users can click to inspect node state or the contents of messages and timeouts. Enabling

| Server to node messages | |
|---|---|
| `start` | Start the node |
| `timeout(type, body)` | Deliver a timeout |
| `message(from, type, body)` | Deliver a message |
| **Node to server messages** | |
| `register(name)` | Register a node |
| `response(state, messages, timeouts, cleared)` | Response to any event |

Figure 3.3: The Oddity API. Oddity can debug systems that implement a simple, JSON-based message API. Once a system node registers with the server, it responds to each message (including the start message, which is sent at the beginning of a debugging session and after a reset) with its updated state, sent messages, and set and cleared timeouts.

detailed inspection is crucial for a debugging interface, since engineers use this information to decide which message or timeout should be delivered next. Oddity supports time travel debugging, allowing engineers to navigate to any previously explored state and explore a branching execution history without starting from scratch.

Space-time diagrams present a whole execution of a system at once, allowing a developer to take a broader view; however, they are likely not appropriate as an interface for an interactive debugger (for instance, there is no standard way of displaying in-flight messages). Oddity users may find space-time diagrams useful, however, for history navigation or as a summary. Oddity has prototype support for space-time diagrams, allowing a user to view a space-time diagram of the current execution. These diagrams, however, are static and not interactive. In future work, these could be integrated more completely into the Oddity interface.

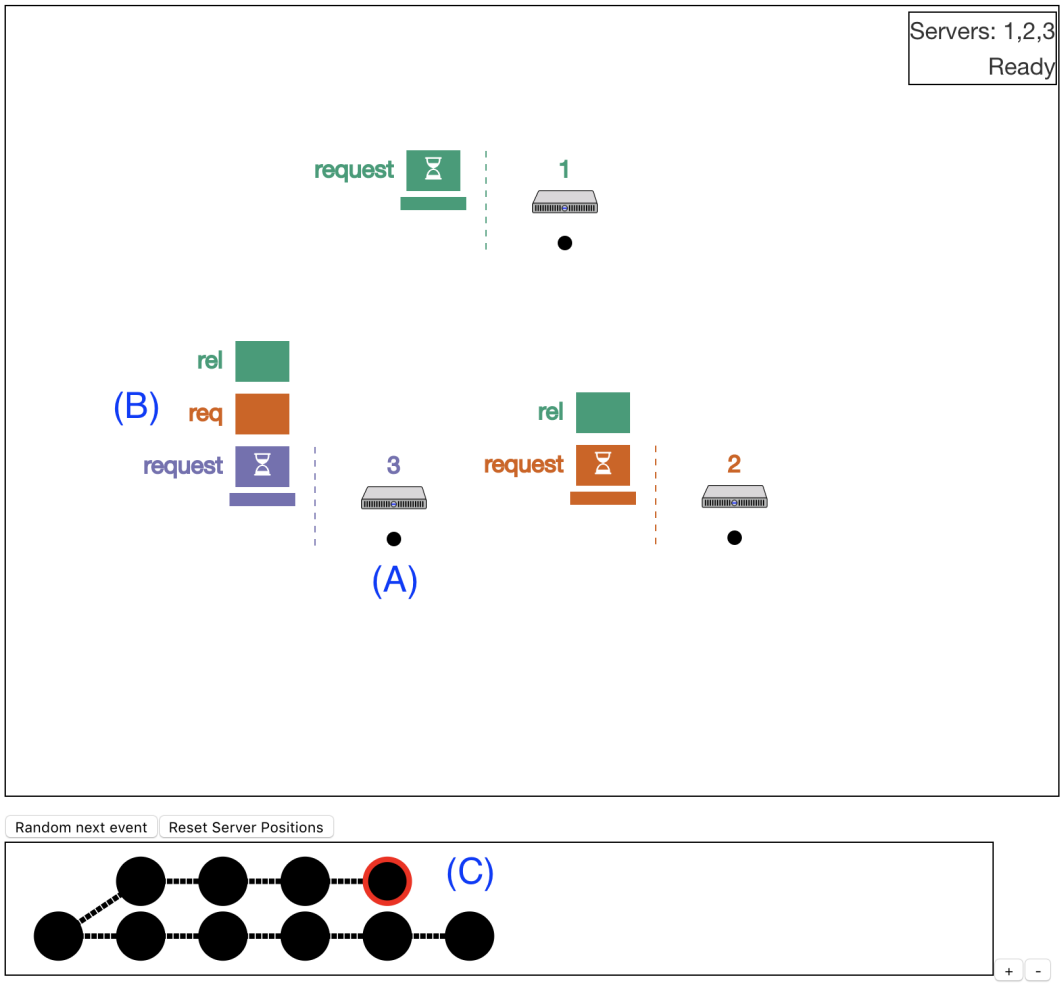


Figure 3.4: The debugger window. Each node (A) is displayed, along with an inbox (B) of messages and timeouts waiting to be delivered at that node. The user can control delivery by clicking on timeouts and messages, and can also inspect the contents of any message or timeout or the state at any node. Using the branching history view (C), the user can navigate the states of the system they have explored. The user can reset the debugger to a previous state by clicking on it; this resets the system to that state so that the user can explore further from there.

**Application-agnostic interface** Oddity's interface is designed to be *application-agnostic*, and can be used for any system where a set of nodes communicating over the network. System-specific visualizations have advantages: they can be clearer, since they can exploit specific visualization techniques that do not generalize. For instance, the Raft visualization included with Runway [48] displays a graphical representation of each node's operation log. In the future, Oddity may be able to provide the best of both worlds: the default system-agnostic visualization could be supplemented by optional, user-supplied system-specific visualizations written against a drawing API and given access to Oddity's view of the system's state.

**Highlighting asynchrony** IN a distributed system, different nodes run at different speeds and messages can be arbitrarily delayed by the network. While messages are often thought of as moving through the network over time to their destination, Oddity does not represent them this way. Instead, messages are immediately transferred to the receiver's inbox and can then be delayed for an arbitrary amount of time (or dropped), under user control. Oddity's display encourages users to ignore wall-clock time in thinking about distributed systems correctness, and instead think about correctness in the face of all possible event orders.

A downside of Oddity's time-oblivious approach is that it may obscure performance considerations; Oddity is unlikely to be useful, for instance, in determining whether a timeout value is too long and likely to cause delays in system execution. Debugging such problems via profiling is a large research area on its own [22, 30, 57].

## *3.3 Oddity implementation*

The research prototype of the Oddity debugger is implemented in approximately 1400 lines of Clojurescript (for the browser-based frontend) and 500 lines of Clojure (for the backend). Interface components are rendered via SVG. The frontend uses a Websocket to communicate with the backend, which communicates with nodes over TCP.

# Chapter 4
# DEBUGGING DISTRIBUTED SYSTEMS WITH ODDITY

As discussed in Chapter 3, Oddity communicates with distributed systems via a simple event-based API (see Figure 3.3). A developer who wishes to debug a system using Oddity must ensure that their system can be controlled via this API. Oddity provides several mechanisms for doing so, depending on the complexity of the system and the developer's priorities. The developer can either (1) replace the system's event loop with one that communicates with Oddity, (2) write an executable model of the system, or (3) use an adapter to automatically translate the system to Oddity's API. The rest of this chapter discusses each method in more detail.

## *4.1 Replacing the event loop*

Some distributed system nodes (for instance, Verdi [58] or Akka [1] systems) are structured as an event loop that waits for network messages or timeouts and then calls deterministic event handlers that update the node's state and produce new messages and timeouts. To integrate such a system into Oddity, the developer must replace the system's event loop with one that communicates with Oddity's virtual network (using the Oddity API) rather than the real network. Rather than blocking on a network message or a timeout, the system's event loop will block on a message from Oddity and then call the correct event handler. For instance, if the message from Oddity is a 'timeout' message (see Figure 3.3) the event loop will call the system's timeout handler.

We have developed an Oddity event loop for the Verdi [58] formal verification framework, and have a version of the `vard` key-value store one can debug using Oddity (which requires recompiling `vard` and linking against the Oddity event loop instead of Verdi's

default event loop). This enables developers to debug Verdi systems using Oddity. Similar event loops could be developed for other event handler-based frameworks, including actor model [8] frameworks such as Akka [1].

### *4.2 Writing an executable model*

If a system is not structured as an event loop calling deterministic handlers—for instance, if system nodes use a model where multiple threads block on remote procedure calls to other nodes—it can still be debugged with Oddity. One option is to write an *executable model* of the distributed system. Executable models are high level versions of a system implemented against the Oddity API (and structured as event loops with deterministic handlers) that are designed to reflect a system's protocol-level behavior. They may elide implementation details such as persistence. Similar models have been used to find protocol bugs in industry; indeed, Newcombe et al. [45] find that developers are sometimes able to catch bugs just by creating such models.

In order to make it easier for developers to write executable models, Oddity includes two "shims"—one for Java and one for Python—against which models are written. Developers write event handlers for the `start`, `timeout`, and `message` commands from the Oddity API (see Figure 3.3); the shim handles communication with Oddity. As discussed in Chapter 6, the Java shim has been used by hundreds of students in distributed systems courses to implement executable models as lab assignments. We have used the Python shim to implement several executable models, including the Raft consensus protocol [49] and the Lamport mutual exclusion algorithm [35].

### *4.3 Using the MajorTom adapter*

The executable model approach has two significant drawbacks. First, it imposes extra work on developers: in addition to developing their system itself, they must create models. Second, there is a risk that the model does not correspond exactly to the system

itself. Indeed, it is likely undesirable to *exactly* model the system–for instance, it is likely not useful to model very low-level details such as unexpected `malloc` return values. It is also possible, however, that the model fails to correctly capture the high-level operation of the system. Perhaps the system implements an optimization (correct or not) that the model does not, or perhaps the system contains a logic bug that is not reflected in the model. In these cases, debugging the executable model using Oddity yields an imperfect understanding of the system's operation. Closing these gaps requires the ability to directly debug the execution of real distributed systems using Oddity.

MajorTom [1] is an adapter between distributed system implementations and the Oddity API. MajorTom responds to events–messages, timeouts, and the special `start` event–delivered by the Oddity debugger by delivering the event to the system and tracking all of the messages and timeouts the system generates, then returning these to the backend. Message sends over the network generate messages, while sleeps and system calls with intrinsic timeouts (such as `select` and `epoll`) generate timeouts. Once the system makes a blocking call, such as a receive or a sleep, the generated messages and timeouts are returned to the server.

MajorTom needs to track the *distributed-systems-related* system calls made by the target system. These include networking, filesystem access, timing, and threading APIs. For most system calls, MajorTom allows the operating system to handle the call as normal. For example, the `sbrk` system call, which extends a process's address space, is allowed to pass directly through to the kernel. For some calls, MajorTom allows the call to proceed as normal but records some information. For instance, for the `bind` system call, which binds a socket so a particular address, MajorTom needs to record the address in order to determine to which node a given message is being sent, but doesn't otherwise need to interfere with the execution of the call. A third type of system call needs to be entirely replaced. For instance, MajorTom cannot allow UDP `send` or `recv` system calls

[1] Major Tom is the protagonist of David Bowie's "Space Oddity"

to go to the operating system, since Oddity needs to be able to control the order in which messages are delivered.

MajorTom supports multi-threaded nodes. Regardless of how many threads it is running, each process in the system corresponds to exactly one node in the display. If a thread is spawned in response to an event, MajorTom allows the spawned thread to execute until it blocks on a receive, a timeout, or a synchronization primitive. It then allows the *spawning* thread to execute, then returns all of the generates messages and timeouts to the Oddity server. Multithreaded programs must be correct given arbitrary thread interleavings, since any thread can be scheduled at any time. One common type of bug in multithreaded programs is the *data race*: when one thread writes to a memory location, another thread reads from the same location, and there is no synchronization between them. These bugs can lead to unpredictable results, especially on complex modern memory architectures. Data race detection, both static and dynamic, has been the subject of a huge amount of research [54, 24]; we consider it to be an orthogonal problem. Since, as discussed below, MajorTom schedules only one thread at a time and runs threads until they block, it will never discover a data race.

In order to obtain information about the types and contents of messages and timeouts, as well as node state, MajorTom includes a debugging information library. Developers link their system against the library and can then annotate message sends or sleeps with information that will then be displayed in Oddity's interface. For local state information, developers can provide a function that returns a representation of the node's state. MajorTom calls this function after every event is executed so that Oddity can display the updated node state.

### *4.4 MajorTom example*

This section illustrates the operation of MajorTom by way of a simple example that exercises some of MajorTom's core features: execution control, multithreading support, and annotation. As shown in Figure 4.1, the system consists of two nodes, `pinger`

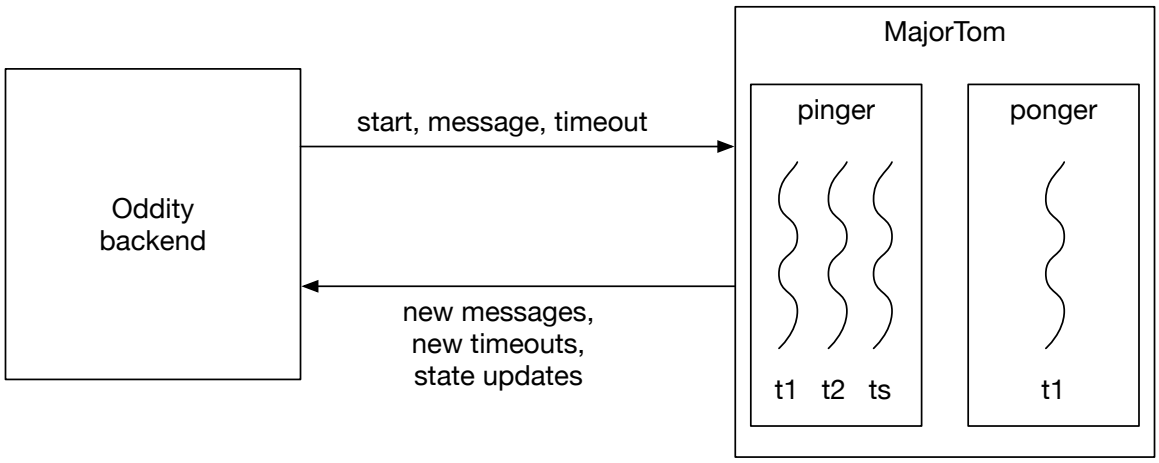


Figure 4.1: The example system. The Oddity backend communicates with MajorTom via the Oddity API. MajorTom executes both of the logical nodes in the system, each in its own process. `pinger` has two application threads–the main thread `t1` and the background thread `t2`. Since it registers a state callback, it also has a thread `ts`, spawned by the annotation library, that repeatedly calls the state callback. MajorTom always wakes `ts` immediately before it returns to Oddity to get the latest logical state. Since `ponger` did not register a state callback and doesn't have a background thread, it only has one thread of execution.

and `ponger`; the C source code for these nodes is shown in Figure 4.2 and Figure 4.3, respectively. The two nodes exchange `ping` and `pong` messages via UDP. `pinger` also runs a background thread which periodically wakes up, sends an extra `ping` message to the other node, and then returns to sleep.

In order to debug this system using Oddity and MajorTom, we first start Oddity. We then start MajorTom with a configuration file with information about how to start the binary corresponding to each node (in this case, `./pinger` and `./ponger`, respectively). Once MajorTom is registered with Oddity, the developer can instruct it to start the system.

```
1  int pings_sent;
2
3  void state_function() {
4    int_field("pings_sent", pings_sent);
5    str_field("name", "pinger");
6  }
7
8  void *background(void *arg) {
9    int secs = *(int*)arg;
10   while(1) {
11     annotate_timeout("Background thread");
12     int_field("seconds", secs);
13     sleep(secs);
14     annotate_message("ping");
15     sendto(sockfd, "ping", 5, 0, ...);
16     pings_sent++;
17   }
18 }
19
20 int main(int argc, char** argv) {
21   char buf[5];
22   int pongs_received = 0;
23   pings_sent = 0;
24   register_state_function(state_function);
25
26   sockfd = socket(...);
27   bind(sockfd, ...);
28
29   annotate_timeout("Start");
30   sleep(5);
31   pthread_t tid;
32   int secs = 5;
33   pthread_create(&tid, NULL, background, &secs);
34   while(1) {
35     annotate_message("ping");
36     sendto(sockfd, "ping", 5, 0, <address>);
37     pings_sent++;
38     recvfrom(sockfd, buf, 5, 0, ...);
39     pongs_received++;
40   }
41 }
```

Figure 4.2: C source code for the `pinger` node. At a high level, the node runs a loop sending `ping` messages to the `ponger` and receiving `pong` messages in response. It also runs a background thread that periodically sends additional `ping` messages. All of its messages and timeouts are annotated, and it registers `state_function` with MajorTom as a callback that reports its logical state.

```
1  int pongs_sent = 0;
2  int pings_received = 0;
3
4  int main(int argc, char** argv) {
5    int sockfd;
6    char buf[5];
7
8    // make a socket:
9    sockfd = socket(...);
10
11   // bind it to the port we passed in to getaddrinfo():
12   bind(sockfd, meres->ai_addr, meres->ai_addrlen);
13
14   struct sockaddr from;
15   socklen_t fromsize = sizeof(from);
16
17   while(1) {
18     recvfrom(sockfd, buf, 5, 0, ...);
19     pings_received++;
20     sendto(sockfd, "pong", 5, 0, ...);
21     pongs_sent++;
22   }
23 }
```

Figure 4.3: C source code for the `ponger` node. At a high level, the node runs a loop receiving `pong` messages from the `pinger` and sending `pong` messages in response. It does not annotate its messages, and it does not register a state callback.

When it receives the special `start` message from Oddity, MajorTom starts each program in order. MajorTom uses `ptrace` to intercept each syscall when it is executed. When it starts `pinger`, it first intercepts a large number of syscalls made by the C runtime: various shared libraries are loaded, the process's address space is set up, etc. The program then starts executing.

The first call `pinger` makes is to `register_state_function` on line 24. `register_state_function` is implemented in MajorTom's debugging information library, which was linked with `pinger` at compile-time. `register_state_function` spawns a new thread, which first registers itself with MajorTom as the node's state callback thread (by calling a special, MajorTom-specific system call) and then loops forever,

repeatedly calling `state_function`. MajorTom allows the thread to spawn, notes that this is the special state callback thread, and then continues executing the main thread.

The next system call is to `socket` on line 27, which creates a UDP socket. MajorTom allows this call to go through to the operating system, and records the file descriptor that the operating system returns if it succeeds. When the program calls `bind` on the same file descriptor on line 28, MajorTom records the address that the program passes. When packets are subsequently sent to that address (e.g., by `ponger`), MajorTom can determine that they are being sent to `pinger`, enabling Oddity to display them in `pinger`'s inbox.

On line 29, `pinger` calls `annotate_timeout`, another function provided by the debugging information library. `annotate_timeout` calls a MajorTom-specific system call, passing its argument. MajorTom uses this argument as the displayed type of the next timeout the node generates. This timeout is then generated by the call to `sleep` on line 30. MajorTom prevents the operating system from handling the call, blocks the thread, and records the new timeout. MajorTom then wakes the state callback thread. It calls `int_field` and `string_field`, both of which are MajorTom-specific system calls that add fields to the current message, timeout, or logical state (these fields are then displayed using Oddity's state inspection facility).

MajorTom then starts `ponger`. The calls to `socket` and `bind` are handled identically to those from `pinger`. When `ponger` calls `recvfrom` on line 18, MajorTom prevents the call from reaching the operating system and blocks the thread from executing further.

Having started both of the processes in the system, MajorTom returns control to Oddity. In its response to Oddity it includes the new timeout set at `pinger`, with type `Start` and an empty body. The resulting Oddity display is shown in Figure 4.4.

If we deliver the "Start" timeout, Oddity sends it to MajorTom. MajorTom then wakes `pinger`'s main thread. Its next system call is `pthread_create` on line 33. MajorTom allows the system call to go through and switches to running the resulting thread, which executes the `background` function on line 8. This function annotates a timeout with a type (`Background thread`) and an integer field (the `secs` argument, 5), then calls


Figure 4.4: The state of the example system after both nodes have been started.

`sleep`. MajorTom records the timeout, then switches back to executing the main thread. The main thread annotates a message with type `ping`, then calls `sendto` on its UDP socket; it passes the address of the socket bound by `ponger`. Since MajorTom recorded this address when `ponger` called `bind`, it records a message with type `ping` addressed to `ponger`. It blocks the send from actually going through and records the sent data, then continues executing `pinger`.


Figure 4.5: The state of the example system after the `Start` timeout is delivered to `pinger`.

The next call is to `recvfrom`. Here, MajorTom blocks the thread while recording that if a message is delivered to the corresponding address, it should wake again. MajorTom again wakes up the state callback thread to get the updated logical state, then returns the new timeout (from the background thread) and the new message (from the main thread, addressed to `ponger`) to Oddity. Now that the `Start` timeout has been delivered, the

system's state is as shown in Figure 4.5.


Figure 4.6: The state of the example system after the `Start` timeout is delivered to `pinger`.

We now have a choice: we can deliver either the `Background thread` timeout to `pinger`, which will wake the background thread and send an additional `ping` to `ponger`, or deliver the `ping` already in the network. We have already seen all of the operations necessary for the former: MajorTom wakes the background thread, records the message and the new timeout, wakes the state callback thread, and then returns to Oddity. For the latter, MajorTom wakes `ponger`'s main thread (since it is blocked on a `recvfrom` on a socket whose bound address corresponds to the message's destination address; see line 18). MajorTom copies the sent data into the buffer `buf` and allows `ponger` to continue executing. Note that for sends and receives (as well as sleeps) MajorTom never actually allows the operating system to handle the calls. The `ponger` then executes the `sendto` on line 20; MajorTom records the message. It is not annotated, so MajorTom cannot send any information to Oddity about its type or body. `ponger` then blocks again on the `recvfrom` on line 18, and MajorTom returns the new message to Oddity. Once it returns, the Oddity display is as shown in Figure 4.6.

Oddity implements backtracking by restarting the system and replaying the same sequence of events. This works unmodified in MajorTom. The only challenge is correctly restarting processes and restoring the filesystem state. When it receives a `start`

command from Oddity, MajorTom first kills the processes corresponding to nodes in the system. It then cleans up the filesystem by restoring all files the processes might have written to their state from before the system started executing.

## *4.5 MajorTom implementation*

MajorTom is implemented for Linux x86-64 binaries using the `ptrace` facility, which is designed for use by interactive debuggers and allows system calls to be intercepted. MajorTom starts the system's processes in response to the `start` message from Oddity. It uses `ptrace` to stop each process at entrances and exits from syscalls, modify the arguments and results, and selectively block calls. MajorTom is implemented in approximately 4000 lines of Rust code.

Implementing the Linux system call interface in its entirety would amount to re-implementing Linux itself. Fortunately, MajorTom can delegate to the Linux kernel for much of its functionality; MajorTom only interposes on system calls that involve persistence, timing, network communication, or multithreading. The rest of this section details MajorTom's implementation of each of these components of the Linux system call interface, as well as its implementation of annotation.

### *4.5.1 Persistence*

Oddity supports navigating a branching execution history. In order to restore the system to a previously-explored state, Oddity first restarts the system and then replays the sequence of events that led to that state. This means that the system cannot be allowed to persist state across restarts—this would represent a source of non-determinism outside Oddity's control. MajorTom therefore interposes on filesystem operations made by the target system. Whenever the system opens a file for writing, MajorTom records a snapshot of the file—or, if it is being newly created, records that the file did not previously exist. When the system is restarted, MajorTom restores these snapshots, writing back the

previous contents of files and potentially deleting newly-created files and directories.

### 4.5.2 *Timing*

Real time represents both a source of non-determinism that MajorTom must control and an input to the system in the form of timeouts. Nodes running under MajorTom are prevented from accessing the system's clock; instead, calls to `gettimeofday()` and similar APIs access a virtual clock MajorTom maintains for each node. This virtual clock is advanced when a node receives a timeout. Each timeout carries an absolute time $t$; when the timeout is delivered to a node, MajorTom ensures that nodes virtual clock is at least at $t$. This is then reflected in subsequent calls to `gettimeofday`, etc.

Our initial MajorTom prototype did not support a virtual clock mechanism: calls to `gettimeofday` always resulted in the same time (midnight on January 1 1970, the start of the Unix epoch). We found that this resulted in confusing behavior by systems: since many systems are built to account for spurious wakeups during sleeps, they check to see whether the correct amount of time has actually elapsed after waking from a timeout. In order to ensure that such systems do not immediately block again, we implemented the virtual clock.

Another approach to the same problem is the one taken by the MoDist model checker [59]. MoDist implements a simple static analysis to detect cases where a system branches on the system time, and ensures that its model checker executes both sides of the branch. This has the benefit of detecting when a system handles spurious wake-ups incorrectly (an intra-node bug).

### 4.5.3 *Network communication*

MajorTom supports both UDP and TCP sockets communicating using IPV4 addresses (IPV6 support would require only minor engineering changes). Since UDP is a connection-less, message-oriented protocol, MajorTom's UDP support is fairly simple.

When a node calls `sendto()` on a UDP socket, MajorTom reads the sent data and the target address from the node's memory, finds the node corresponding to the target address, and then sends an Oddity-level message. When a node calls `recvfrom` it blocks until a UDP message arrives for its bound address; MajorTom then writes the received data and the remote address into the node's memory and lets the call return.

TCP support is more complex, since TCP is a connection-oriented protocol. The general approach is to allow TCP sends and receives to go through the operating system, but to proxy connections through sockets controlled by MajorTom. MajorTom can then control when data are delivered. To Oddity, MajorTom represents all consecutive TCP sends as one message (unless instructed otherwise by annotations). TCP includes a "handshake" protocol to establish connections. Rather than representing each part of the handshake protocol as an individual message in Oddity, MajorTom simplifies the protocol to an initial `Tcp-Connect` message followed by a `Tcp-Ack` message. Either message can be delivered or dropped, as normal.

Modern distributed systems frequently rely on non-blocking I/O operations. MajorTom supports these operations. Rather than requiring that a node is waiting for a message when one is delivered, Majortom tracks a queue of incoming messages for each socket. Socket reads pull from the queue if a message is available; otherwise, they either block or return an appropriate error code.

Rather than dedicate a thread per client, modern systems often service many clients from the same thread, blocking on a set of sockets until one can be read. Linux provides the `epoll` family of system calls to efficiently poll a set of sockets. Since calls to `epoll` block in the kernel, MajorTom must re-implement `epoll`'s functionality.

### *4.5.4 Multithreading*

Most modern distributed systems run multiple threads of execution at each node. MajorTom interposes on each thread's execution, so it must track thread creation. MajorTom

tracks each thread separately, with shared state between threads for file handles, virtual time, etc. MajorTom must avoid waiting on blocked threads, so it must handle synchronization primitives such as mutexes and condition variables.

Linux provides one thread synchronization primitive: the `futex` (fast userspace mutex). MajorTom implements all of the futex operations used by the standard `pthreads` threading library. In this case it was necessary to re-implement a large piece of operating system functionality in order to avoid blocking in the kernel.

### *4.5.5 Debugging information library*

MajorTom obtains debugging information via a separate, optional library. It contains functions for annotating messages and timeouts, and for registering a node's state callback function. These are all implemented via fake system calls (whose system call numbers are unused by the operating system), which MajorTom then manually handles. The implementation of the state callback function is slightly complex: because `ptrace` does not expose a method of calling a function in the address space of a traced process, when a node registers a state callback function the debugging information library spawns a new thread that repeatedly calls the callback and then calls out to MajorTom. MajorTom wakes this thread after each event is processed and sends the updated state to the server. Many distributed systems use standardized serialization formats such as protocol buffers [4] or JSON for messages. Oddity includes special support for protocol buffers, enabling developers to easily annotate their protocol buffer-based messages.

# Chapter 5
# THE ODDITY MODEL CHECKER

The Oddity prototype discussed in the previous sections is useful (see Chapter 6) but has some limitations. One is that the original prototype has no provisions for automatically executing a system. When debugging, users must manually deliver each timeout or message. If a user is not sure which sequence of messages and timeout deliveries will lead to an interesting state, they are left manually exploring paths. Even if a user does know which sequence of events leads to a desired state, long and predictable event sequences can be tedious to replicate.

In order enable this kind of automatic execution, we developed a new model checker and integrated it into Oddity. A user can specify a predicate describing a state they want to find–for instance, in Raft, a state where a particular node has won an election and become the leader. The Oddity model checker then runs, searching the space of possible executions for one that leads to a state matching the user's predicate. The user can start the model checker from any state; the model checker will only consider executions starting from that state.

## *5.1 Model checker interface*

Oddity's model checker is presented to users as a "Run until" command: Oddity runs the system until it finds a state matching a desired predicate. The model checker allows predicates of the form $node.path = value$, where *node* is one of the nodes in the system, *path* is a period-separated path to a state variable, and *value* is a string or a number. More general predicates would be trivial to implement, but we anticipate that this form, specifying the value of a single state component of a single node, will suffice for the

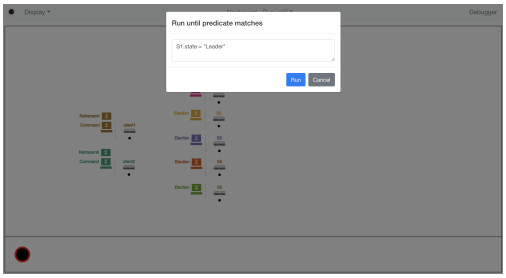


Figure 5.1: The interface for the Oddity model checker.

majority of use cases. The model checker's graphical interface is shown in Figure 5.1. When the user clicks "Run," the model checker starts searching for a matching state.

If the model checker finds such a state, the sequence of events leading to it is displayed in the history view just as if the user had executed the sequence by hand. The current state is advanced to the state matching the predicate, and control is returned to the user. If no such state is found within a timeout (10 seconds, by default) the model checker ends the search and control is returned to the user.

## 5.2 *Model checker architecture*

Model checking [16] is a set of techniques for verifying finite-state systems or finite subsets of infinite-state systems. Model checking ensures that a desired property holds in a system by checking that the property holds on all possible executions of the system. Model checking is divided into two types: symbolic model checking and explicit-state model checking. In symbolic model checking, the program is translated into a logical formula; this formula is then checked for validity. Explicit-state model checking can check either programs or high-level program models. In either case, an explicit-state model checker exhaustively searches the program's state space (the space of all possible combinations of variable values) by controlling the program's inputs and any non-determinism (e.g., scheduling decisions). For programs with infinite state spaces, explicit-state model checkers can explore all possible states within particular bounds—either on components

of the state (e.g., a particular variable's values could be artificially bounded to a given range) or on the number of state transitions the program can take. There is a long line of work on model checking distributed systems [32, 59, 15, 28, 38].

Oddity's model checker is explicit-state and includes a depth bound. Oddity's It is implemented as part of its backend (see Chapter 3). A model checking request from the user consists of a sequence of events $P$ (i.e., the sequence leading to the system's current state) and a predicate $\phi$ over the states of nodes in the system. The model checker's goal is to find a sequence of events $S$ such that $P$ is a prefix of $S$ and that when the sequence $S$ is run starting in the system's initial state the resulting state satisfies $\phi$. The model checker is implemented as an iterative bounded depth-first search: it first explores all execution traces with length equal to an initial depth bound $d$ (6 by default), then all traces of length $2d$, then $3d$, etc., until it times out (this is done so that it is likely to find useful, short traces as soon as possible, while still allowing it to find longer traces). It explores a trace by first resetting the system, then executing the prefix $P$, then executing the trace, then checking to see whether the resulting state matches the prefix.

Our intuition is that messages are more likely than timeouts to advance the state of the system in "interesting" ways (since timeouts often simply result in a re-sent message), and that events that directly interact with the node mentioned in $\phi$ are more likely to result in interesting states at that node. In order to find interesting states more quickly, the model checker employs a heuristic to determine the order in which it adds each possible action to a trace. In order, the model checker prefers:

1. Messages to the node mentioned in $\phi$
2. Messages from the node mentioned in $\phi$
3. Other messages
4. timeouts to the node mentioned in $\phi$

5. Other timeouts

# Chapter 6
# EVALUATION

Oddity is intended to help developers find and fix bugs in their distributed systems. To that end it includes a number of features: message and timeout inspection, backtracking, model checking, system call interposition, etc. In evaluating Oddity, we have attempted to determine whether it succeeds in its goal of helping developers, as well as whether each of its features is necessary to achieve that goal. As such, this section attempts to answer several research questions:

**RQ 1** Is Oddity useful?

**RQ 2** Are Oddity's features useful?

- **RQ 2a** Is message and timeout inspection useful?
- **RQ 2b** Is backtracking useful?
- **RQ 2c** Is model checking useful?

**RQ 3** Can system call interposition work on practical systems?

We evaluated RQ1 and RQ2 by deploying Oddity to two distributed systems courses, in the Spring of 2018 and the Spring of 2019. We studied student usage of Oddity via opt-in telemetry and optional qualitative surveys. The model checker had not yet been developed as of Spring of 2018, so we use the data from that class to answer RQ's 1, 2a, and 2b. We use the data from the Spring 2019 deployment to answer RQ 2c. These evaluations are described in Sections 6.1 and 6.2.

We evaluated RQ 3 by implementing system call interposition for two systems: LogCabin [3], the reference implementation of the Raft consensus protocol, and Redis [5], an in-memory database widely deployed for caching, user sessions, and queueing. We provide a qualitative evaluation of the effort necessary to get Oddity working on each system. This evaluation is described in Section 6.3.

### *6.1 Classroom evaluation of the original Oddity prototype*

In the Spring of 2018, we deployed Oddity to a 180-student undergraduate-level distributed systems class (CSE 452 at the University of Washington). Oddity was given to students as part of DSLabs [43], the lab framework they used to do their homework assignments. The labs come with extensive test suites, and include a model-checker. Students could run Oddity in two modes: they could start their system and explore from the beginning of time, or run Oddity on a counterexample trace produced by the model-checker when it detected an invariant violation. Note that the model checker included in DSLabs is distinct from the Oddity model checker: it is run outside of Oddity, as part of the test suites included with the labs.

We had two goals in studying students' experience with Oddity. One was to determine whether Oddity's features are useful. The other was to examine student behavior when given access to an interactive debugger for distributed systems, in line with previous work that examines student usage of traditional step-through debuggers [42] and developer usage of trace visualization tools [18]. We hope that our experiences can inform future work in the same area.

We studied student experiences with Oddity in several ways. We instrumented the Oddity interface in order to track users' clicks on various interface elements in order to see how they interacted with the system (this feature was only enabled if students opted into it). We sent out an optional survey to students after they completed the first major lab assignment, a primary-backup-based key-value store. The survey is shown in Figure 6.1. We also informally discussed Oddity with students, recording anecdotes

about their usage of the system on the primary-backup lab as well as the next lab, a Paxos-based key-value store [37].

We found that 74.5% of Oddity runs started from the beginning of time, as opposed to from a model-checking trace. In these runs, users explored an average of 37.3 states per run, with a median of 23 states per run. From this we can conclude that students did use the debugger for exploratory testing. We received some survey data to suggest that students were able to explore edge cases using this mode. One student said that *"It was useful for one bug where I found out there was unexpected behavior from the [view server] when both the primary and backup timed out at the same time."* This indicates that students used Oddity to explore edge case behavior. We also received an anecdote from a student about a bug in which their Paxos implementation sent redundant messages under certain conditions (specifically: when a proposer received more than a majority of replies to its "prepare" messages, it ended up sending extra "accept" messages). The student did not suspect the existence of this bug before noticing it in Oddity, and believed they would not have found the bug at all without Oddity (the provided test suite did not test for the presence of these extra messages). Without an interactive debugger that can control message and timeout ordering, exploratory testing of distributed systems is tedious, and the usage tracking and survey results indicate that students find exploratory testing using Oddity very useful.

A number of students said that they did not explore their systems starting from the beginning because they only debugged their system when a test case from the provided test suite failed. Our results may be biased as a result of our setting: with an extensive test suite, some students may not have felt a need to understand their system behavior independently of its behavior on the tests. It is possible that without such an extensive test suite, students would have found it more useful to start their systems in the debugger. On the other hand, our evaluation is of students and not professional developers. The students were learning about the protocols at the same time as they were implementing and debugging them, so it is possible that exploratory testing was a more

compelling option for students than it would be for more experienced developers.

We found that 25.5% of Oddity runs started from a model-checking trace. In response to survey question 2, students reported that Oddity *"helped [them] diagnose [their] handling of state transfer and state transfer acknowledgements"* and that they were able to use it to diagnose a bug in which they *"had some delayed messages arriving and causing problems."* A student reported successfully using Oddity to diagnose a bug in which the system had stopped making any progress after their latest change, which implemented de-duplication of redundant client requests. They stepped through a simple test case and found that servers were never actually responding to clients; they were then able to fix the issue.

When students started their systems from a model-checking trace, 23.6% of those executions explored multiple branches. In those cases, those state graphs branched an average of 1.5 times. From this we can conclude that at least some students explored alternative executions when viewing a model-checking counterexample. In response to survey question 3, some students did report that exploring alternative executions was useful. One student said that *"from the bug where our servers were advancing themselves based on outdated/future view numbers, instead of just from the view server, it helped us see a situation where we could get stuck more frequently waiting for the server to ack a state transfer."* Another reported that the ability to explore alternative executions *"distinctly helped understand what was happening."* We can conclude that the ability to explore alternative executions starting from a model-checking counterexample was useful for some students.

### *6.2 Classroom evaluation of the Oddity model checker*

In the Spring of 2019, we deployed Oddity to a subsequent iteration of CSE 452. We had made a number improvements to Oddity, the largest of which was to add the model checker described in Chapter 5.[1] Our goal in this second deployment was to determine

[1]The other improvements were various user interface tweaks and bug fixes.

1. Did the debugger help you to discover any bugs in your system? Describe one.
2. Did the debugger help you to diagnose any bugs you were already aware of? Describe one.
3. When using the debugger to view a search-test counterexample, did you also explore other executions? Did this help you to understand the counterexamples? Describe an instance of this being useful.
4. Were there any bugs you think you would have found earlier if you had used the debugger? If not, how could the debugger have been more useful to you?
5. Do you have any other feedback about the debugger?

Figure 6.1: The optional survey sent to students after completing a homework assignment. We called tests that used the DSLabs model checker "search tests."

whether the model checker is a useful feature. To that end, we used an optional survey (just as in the previous deployment). The author also provided instruction on how to use Oddity, and included a demonstration of the model checker being used to both (1) easily automate tedious event sequences and (2) check for erroneous behavior. The survey we distributed to students is shown in Figure 6.2.

At a high level, we found that very few students used the Oddity model checker. We believe that this was due to the nature of the debugging students did: since the students in the class had access to an extensive test suite (which itself included model checker-based testing), most students did not see any need for the Oddity model checker's features. One student did report that they were able to use the model checker to skip over tedious steps in an execution, one of the model checker's intended use cases. Further

1. Did the debugger help you to discover any bugs in your system? Describe one.
2. Did the debugger help you to diagnose any bugs you were already aware of? Describe one.
3. Did you use the model checker in order to speed up debugging? Describe this usage.
4. Did you use the model checker to increase your confidence in your system's correctness? Describe this usage.

Figure 6.2: The optional survey sent to students after completing a homework assignment.

study is necessary to determine whether the model checker is of general utility.

### *6.3 Evaluation of MajorTom*

MajorTom is an adapter that allows users to debug arbitrary existing distributed system implementations using Oddity; its implementation is described in Chapter 4. MajorTom works by interposing on the distributed systems related system calls made by processes in the system. Rather than allowing processes to exchange messages directly with each other, MajorTom gives the user control over message delivery (via Oddity).

Fully implementing system call interposition for an arbitrary system would be prohibitively time-consuming; the Linux system call API (our target platform) is quite complex and involves various difficult-to-handle edge cases. We have instead implemented a subset of the API that is sufficient to run some test systems.

We have implemented enough of the Linux system call API to support two target

systems: LogCabin and Redis. LogCabin was used as a target system during the development of MajorTom, so MajorTom's system call support was informed by the system calls required by LogCabin. Redis support was added separately, and we tracked the amount of additional work required to support Redis in order to determine the expected difficulty of supporting new systems. MajorTom is a research prototype, and is unlikely to work in its current form on an arbitrary distributed system without significant changes, either to the system or to MajorTom itself. Our current demonstration systems serve as a proof of concept: it is possible to debug practical distributed systems using Oddity and MajorTom.

### *6.3.1 Supporting LogCabin*

LogCabin [3] is the reference implementation of the Raft consensus protocol. Its source code consists of approximately 100,000 lines of C++. LogCabin nodes communicate with each other and with clients over TCP using protocol buffers [4]. LogCabin nodes use `epoll` (discussed in Chapter 4) to block on multiple sockets and timers. LogCabin nodes run several background threads, used for communication (e.g., sending "heartbeat" messages to other nodes), monitoring, and persistence. These threads are synchronized using the mutexes and condition variables included with `pthreads`.

We have implemented support for *configuration* of a LogCabin cluster. Specifically, a set of LogCabin nodes, as well as a client, can be started using MajorTom. The client is a particular program included with LogCabin that sets up a cluster of Raft nodes, informing each of the existence of the others and ensuring that a leader is elected. A user can step through the reconfiguration process in Oddity, observing each message that nodes exchange. Every message and timeout is annotated (using MajorTom's protocol buffer support), and each LogCabin node's state can be inspected (also using MajorTom's protocol buffer support; every node tracks statistics in a protocol buffer).

This support required the addition of 11 annotations on timeouts and messages, as

well as the addition of a state callback. It also required some minor changes to LogCabin's code. First, we added a timeout to the start of the client process's `main()` method. This was necessary because MajorTom starts nodes in a non-deterministic order. When the client starts, it immediately sends messages to LogCabin nodes. If MajorTom has not yet started those nodes, it does not know which logical node a given address corresponds to. Second, it was necessary to modify a timed conditional wait method to include annotation information in order to ensure every timeout was annotated.

The reconfiguration client is the only client we have debugged using MajorTom. It is possible that other clients would trigger unsupported code paths in LogCabin, but it is unlikely that large changes to MajorTom would be required; MajorTom supports LogCabin's main `epoll` loop, which is the only way that LogCabin nodes interact with the network. It is likely, however, that additional annotations in LogCabin would be necessary to obtain useful debugging information for these other code paths.

### *6.3.2 Supporting Redis*

Redis [5] is a widely-deployed in-memory data structure store. It includes support for eventually-consistent replication and sharding via a subsystem called Redis Cluster. We implemented support for Redis Cluster in MajorTom in order to determine the difficulty of adding support for a new system.

Redis's source consists of approximately 150,000 lines of C code. Like LogCabin, Redis relies on an `epoll`-based event loop. In addition, Redis uses non-blocking socket calls ubiquitously. Adding support for Redis required TODO: **[[N]]** modified lines of code. This consisted of a number of minor changes, such as adding pass-through support for distributed systems-irrelevant system calls (e.g., `prlimit64()`) and translating `read()` and `write()` calls on TCP sockets to `recv()` and `send()` calls. There were also two larger changes. First, Redis Cluster relies on each node generating a unique identifier by reading from the `/dev/urandom` pseudo-random number generator. Ma-

jorTom already supported deterministic reads from `/dev/urandom`, but these reads were the same across different nodes. We fixed this issue by seeding a random number generator with each node's name and then (deterministically and repeatably) using this generator to serve `/dev/urandom` reads. Second, Redis uses non-blocking socket reads ubiquitously. Correctly handling these required adding support for non-blocking sockets, and queues of messages waiting at each such socket, to MajorTom.

We did not annotate Redis with debugging information. Redis uses a custom format for both its internal and external messages. The simplest way to obtain debugging information for messages would be to write a translator (internal to Redis) from this format to the annotation format expected by MajorTom. This could then be used on every message send. Redis nodes have only one "tick" timeout; annotating it would not be difficult. Annotating Redis's node state would require writing a state function that read user-relevant parts of Redis's state.

# Chapter 7
# RELATED WORK

The notorious difficulty of developing correct distributed systems, and of discovering and diagnosing bugs in existing distributed systems, has inspired a large amount of research on tools and techniques to help developers tackle these challenging problems. We categorize previous work based on several attributes:

**Models vs. implementations** Some tools help developers reason about the high-level protocols and design of their systems by enabling them to develop, test, simulate, and check high-level models of their distributed systems. Other tools operate on concrete system implementations. Oddity can be used on both executable system models and on system implementations.

**Online vs. post-mortem** Some tools involve running the system (or a high-level model) in order to find or diagnose bugs. Others are designed for post-mortem analysis: once a bug is discovered, these tools help the developer understand logs produced by the system. Oddity is designed to be used online, but can also be used to visualize logs.

**Interface type** Most tools do not include graphical interfaces. Nonetheless, there is a long history of distributed systems visualization which Oddity draws on for its interface

**Techniques used** Some techniques are shared in common across many tools. In particular, many tools (including Oddity) apply model checking, either to high-level

models or concrete implementations. We therefore discuss model checking in some detail below.

### *7.1 Debugging system models*

Distributed system implementations can be quite complex. Rather than reasoning directly about implementations, several systems require developers to create high-level models of the algorithms and protocols used in their systems, usually in languages designed for modeling and reasoning. Properties of these models can then be tested, simulated, model-checked, and even formally proved correct.

We discuss two such tools, TLA+ and Runway, below. Spin [28], an explicit-state model checker, as well as symbolic model checkers such as the Alloy Analyzer [29] and NuSMV [15], have also been used to develop and check very high-level models of distributed systems in similar ways. These systems differ mostly in their input languages—Spin's and NuSMV's are based on Linear Temporal Logic, while Alloy's is relational. Molly [9] efficiently checks system models for fault-tolerance bugs using SAT solvers and data lineage tracking.

**TLA+ and TLC** TLA+ [38] is a specification language based on the Temporal Logic of Actions (TLA) [36]. TLA is a logic, based on modal logic and set theory, developed by Leslie Lamport for reasoning about systems in general and distributed systems in particular. A system in TLA consists of a number of global variables, an initial state predicate, and a set of *actions*. An action is a logical formula over the variables in a system where some of the variables are primed. For instance, is $A$ is an integer variable, $A < 10 \wedge A' = A + 1$ is a valid TLA action. An action represents a state transition; the primed version of a variable describes the variable in the subsequent state of the system.

TLA+ is most commonly used in conjunction with TLC, a model checker for system specifications written in TLA+. TLC is an explicit-state model checker for TLA+; starting from the initial state of a system, it enumerates all possible reachable states and can

check invariants specified in TLA+. Since the model checker explores the whole state space, it can only be used on finite models; users thus generally manually finitize their models by bounding aspects of the system state. TLA+ and TLC have been effective in industry. Newcombe et al. [45] used TLA+ (and a simple imperative language, PlusCal, that compiles to TLA+) to model several complex system components inside Amazon Web Services; they found that they were able to find bugs at the design stage rather than in testing or, worse, in production.

Like TLC, Oddity can be used to debug and understand system models. Unlike in TLA+, Oddity's models are executable code, and Oddity enables the user to step through an execution graphically. For some systems, TLA+ allows modeling at a higher level of abstraction—for instance, it may sometimes be desirable to elide explicit message sends and receives in favor of atomic actions. Users have also reported (e.g., in [45]) that the act of formally specifying a system in TLA+ is itself useful for considering edge cases; it is unclear whether developing an executable model for use with Oddity would have the same effect.

**Runway** Runway [48] is a browser-based environment for visualizing and checking models of distributed systems. It consists of a domain-specific programming languages for specifying models of distributed systems, along with an interpreter for this language written in Javascript and an API for extracting values from the interpreter for visualization. Several models and animations have been developed using Runway, including one for the Raft consensus protocol [6].

The Runway language is based on Lamport's Temporal Logic of Actions (TLA) [36]. A Runway program consists of a set of global variables and a number of *rules* (corresponding to actions in TLA); a rule is an atomic transition between states of the system. The Runway language is carefully constructed so that all global variables are bounded: numbers have types guaranteeing that they fall in a particular range, and scalar types must have bounded size. The bounds on global variables means that there are a finite

number of states of the system. There is one unbounded type in the system: `Time`, representing a global clock. The Runway language also includes invariants, which are predicates on the global state of the system. Runway's visualization engine is simple: it requires the user to write a Javascript function that can query the model and writes an SVG document to the browser DOM.

Once a user has developed a model and a visualization, Runway provides several ways to interact with it. The default is to simulate an execution of the model. Runway's simulation engine periodically picks a valid-to-execute rule (a rule is valid if and only if executing it will change the current state; rules that don't change the state will not be executed), runs that rule, and then runs the user's visualization code on the resulting state. The user can change the simulation's speed, and can play or pause it. The user can also take control of the simulation. While the simulation is paused, the user can choose a valid next action to execute. The user can also directly edit the state of the model. Runway also includes a simple model checker that performs a bounded-depth search over states of the system checking for invariant violations.

Oddity's visualization was inspired in part by visualizations developed with Runway. Runway does not include a "default" visualization which can be used for any system; a visualization like Oddity's could be used for that purpose. Runway's focus is more on simulation (animation) than on interactive usage, and the included visualization reflect that–for instance, Runway's Raft visualization shows messages "in transit" between nodes, which makes for a compelling animation but does not lend itself to interactive control. Runway programs are comparable in some ways to the executable system models developed for Oddity, but have to be written in Runway's language–in exchange for which Runway can guarantee finiteness for bounded model checking.

### *7.2 Finding bugs in implementations*

A related line of work focuses on finding bugs in distributed system implementations via techniques such as model checking, fuzz testing, and interactive debugging.

**Mace** Mace [32] is a language extension for C++. Mace uses a compositional event-handler style: a Mace node is implemented as a series of layers, communicating with each other via up-calls and down-calls. The bottom-most layer is a routing layer, implementing communication over the network. Mace also has aspect-based facilities for implementing crosscutting concerns such as assertions over global system state, failure detection, and logging.

MaceMC [32] is an explicit-state model checker for distributed systems implemented in Mace. Unlike most model-checkers, which check safety invariants, MaceMC is targeted at finding liveness violations: specifically, situations in which the system enters a state from which it can never make progress. MaceMC replaces the routing layer at each Mace node, allowing it to control the order in which events happen in the system; it also interposes on sources of non-determinism (e.g., random number generation) at each node. MaceMC finds liveness violations using a combination of bounded depth-first search and random walks through the state space. First, MaceMC finds all states reachable in a finite number of steps. Since the depth is unlikely to be sufficient to allow the system to satisfy its liveness property, MaceMC then starts a random walk from each of these states. If such a random walk fails to reach a live state, MaceMC has found a potential liveness violation, which it reports to the user.

For one of the systems they tested, the MaceMC authors found it necessary to do a *prefix-based* search. The system in question is a distributed hash table with a complex initialization procedure that the model checker had trouble finding on its own. The authors hard-coded this initialization procedure into the search so that each path explored started with the initialization. We believe that Oddity can be used to enable similar techniques more easily—the user can simply click through the initialization procedure rather than hard-coding it.

Mace also includes a non-graphical debugger, MDB, based on the same techniques (controlling event ordering and non-determinism, exploiting atomicity, and replay-based backtracking) as MaceMC. The debugger allows a developer to explore one or more

execution logs and examine the state at each node. The developer can also control which event is to be delivered next, creating a branch off of the trace. The Mace developers found the debugger useful in understanding counterexamples to liveness produced by the model-checker. Oddity, then, is similar to a graphical version of MDB with support for arbitrary systems, not just systems developed using Mace.

**MoDist** MoDist [59] is a *transparent* explicit-state model checker for distributed systems—that is, it is designed to check unmodified application mode. By interposing on distributed systems-related calls made by the application (network, threading, and time-related calls), the MoDist model checker controls the order in which events happen. MoDist can detect application crashes and liveness errors (when MoDist times out waiting for an application), as well as violations of global assertions if the developer is willing to write them.

MoDist's implementation is divided into two parts: an agent library that interposes on system calls made by each application binary and a model checking engine that communicates via RPC with each application binary's agent and controls the order in which events happen. When the system is started, each application binary runs until it issues a system call on which the MoDist agent interposes (thread creation, a read or write from the network, a read from the local clock, or a call to `sleep`). Rather than executing the system call and returning control to the calling thread, the agent instead sends an RPC to the model checking engine indicating which system call was requested, and then blocks waiting for the model checking engine to respond. The model checking engine then decides which available action to execute next, then replies to the RPC from the agent in question. The agent then returns control to the application–either by allowing the system call to go through as normal, or returning an error if told to do so by the model checking action. The model checking engine waits for the agent to block on another system call; if it does not do so within a timeout, MoDist reports a liveness error. At a given time, therefore, only one thread is executing across all application nodes (this

means that MoDist will not detect data races).

MoDist tracks dependencies between available actions: for instance, if one machine reads from a socket, MoDist will never schedule the read before another machine has written to the other side of the socket. It might, however, cause the read to return an error. MoDist injects various failures into the running system, by returning error codes or by hanging up sockets. Since the agent controls the order in which both writes happen, message re-orderings are explored automatically. MoDist also interposes on the application's disk writes and injects realistic failures.

MoDist explores many possible executions, and therefore needs a way of backtracking to execute alternative actions from a given state. Like Mace, MoDist uses replay: since it controls all of the sources of non-determinism in the system, it can restore a previous state by resetting each application binary to its start state, then scheduling the same sequence of actions.

Oddity's mechanism for supporting arbitrary distributed systems is inspired by MoDist. Like Oddity, MoDist interposes on system calls made by the program, giving the model checker control over the ordering of events. Since Oddity gives control over these events to the user rather than to a model-checker (unless, of course, the watchpoints feature is used) Oddity's events are higher-level. For instance, Oddity attempts to coalesce successive writes to a socket into one high-level message send and immediately lets the send go through, only returning control to the user when a machine reads from a socket or waits on a timeout. More details on Oddity's system-call interposition mechanism is in Chapter 6.

**Model checking and Oddity** Human intuition can be used to tune the parameters of a model checker so that it is more likely to find bugs. For instance, MaceMC allows the user to tune the probabilities of various events happening–message deliveries, timeouts firing, failures, etc. It also allows the user to start from a known, hardcoded prefix of events; this is useful if a system has a complex initialization procedure that a model

checker is unlikely to discover. MoDist allows the user to specify a bound on the total number of failures, since many bugs can be triggered with a small finite number of failures and removing the possibility of failure for most actions can drastically reduce the search space. MoDist also has multiple search strategies (random search, breadth-first and depth-first search, and a dynamic partial-order reduction [25] implementation) which the authors manually combined and tuned in order to find systems bugs. We hypothesize that the deep understanding required to use these tools effectively has contributed to their lack of adoption in industry settings.

Graphical interactive debuggers have a complementary set of strengths and weaknesses. Novice developers learn how to use debuggers early on, and stepping through a single execution of a program makes intuitive sense. It is easy to effectively use a debugger even without understanding the debugger's internals. On the other hand, model checkers enable exhaustive search of a large space, enabling users to find bugs on inputs or event sequences they would not otherwise have considered. Oddity attempts to bridge this gap by including a model-checker.

For distributed systems, interactive debugging and model checking rely on similar techniques: both require reproducible control over the execution of a system. In the case of a model-checker, this control is exercised automatically by the search procedure; in the case of interactive debugging, this control is exercised by the user. Performance is more important for a model checker, since it needs to explore as much of the state space as it can as quickly as possible. For an interactive debugger, it is much more important that a user be able to understand the details of a system's behavior and the meaning of each state transition.

**DEMi** DEMi [55] is a fuzz testing tool for distributed systems. In sequential fuzz testing, a program is run on various randomly-generated inputs until an error is triggered or the developer decides to leave well enough alone. In distributed fuzz testing as implemented in DEMi, the "inputs" are a combination of messages from sources external

to the system, scheduling decisions, node reboots, and node startups. DEMi targets the Akka framework [1], and interposes on all inter-actor communication in a manner similar to iDeA (discussed above). DEMi first repeatedly runs the system on long sequences of external and internal events, periodically pausing the system to check for violations of developer-defined predicates. If a violation is found, DEMi then attempts to minimize the execution trace—unlike model checkers, which generally produce relatively short traces since their searches are depth-bounded, traces from fuzz testers can be quite long. DEMi minimizes traces using a combination of delta debugging [60], a known trace-minimization technique, and the partial order reduction. DEMi first minimizes the sequence of external events, then focuses on internal events.

Like the model checkers discussed above, and unlike Oddity, DEMi is designed primarily as a bug-finding tool. Developers *can* find bugs using Oddity, but they can also use it to diagnose bugs or to explore and understand correct system behavior.

**Jepsen** Jepsen [2] is a set of tools for injecting faults into distributed systems executions and checking for violations of externally-specified properties such as linearizability [27]. Jepsen executes system implementations while injecting faults such as network partitions and node failures, then examines the generated executions to find bugs. Jepsen has been used to find data-loss bugs in deployed distributed systems. Like DEMi, Jepsen is a bug-finding tool.

**iDeA** iDeA [41, 51] is a virtual reality (VR)-based debugger for programs written using Akka [1], an actor-model framework for the Scala programming language. In Akka, as in other actor-model libraries and languages, a program consists of a number of "actors" whose behavior is defined by event handlers and local state. Actors only communicate with each other by sending messages; no state is shared between actors. Actors can thus be partitioned among many threads, processes, or machines. Each actor has a mailbox of messages it has received from other actors. When an actor is scheduled, it processes

the next message in its mailbox by running the event handler corresponding to that message's type. This event handler may send new messages to other actors and can update the actor's local state. IDEA consists of two parts: a custom scheduler for Akka programs and a VR interface.

The custom scheduler exposes an API to the VR interface, allowing the user to control the order in which messages are delivered to actors. Internally, the custom scheduler replaces Akka's built-in scheduler. It controls the order in which messages are added to mailboxes–rather than delivering each message immediately, it buffers all messages, informing the VR interface of which messages have been sent and allowing the user to control the delivery order.

The VR interface represents each actor as an object (the specific objects used for each actor can be customized by the user) in a 3D space which the user can explore either by physically walking through it or by teleportation. An actor A is displayed with an arc drawn to actor B when B's mailbox contains a message from A. When the user selects an actor using the VR system's controllers, the interface displays that actor's local state, the number of messages awaiting it, and the next message. The log of events (each event corresponding to delivery of a particular message) is displayed in a textual window.

Timeouts are handled by a special "scheduler" actor (the user must rewrite their code to use this actor if they relied on Akka's native timeout mechanism). Virtual "time" elapses when a special message is delivered to the scheduler actor; this causes it to send timeout messages to other actors in the system as appropriate.

IDEA can be used in several ways. The simplest is to replay a particular, perhaps problematic, logged execution. The user advances the log deterministically, observing how each actor modifies its state and sends messages in order to attempt to diagnose a problem or understand the system's behavior. IDEA can also be used to enforce a particular schedule of the user's choosing on the system. By selecting an actor, the user can enforce that actor as the next to be activated; its event handler will then be run on the next message in its mailbox. In this way, IDEA can be used as an interactive debugger.

Unlike IDEA, which is designed specifically for Akka systems, Oddity can be used with any system (via MajorTom or an executable model). Oddity's visualization is also significantly different from IDEA's; further study is necessary to determine which interface is more effective.

### *7.3 Network Simulation*

Network simulators such as `ns-3` [53] are designed to simulate real-world network topologies, routers, and applications in order to examine their effects on performance and connectivity. Oddity can be viewed as a higher-level, graphical version of such a tool: rather than operating at the level of switches and packets, Oddity assumes full connectivity between nodes (while allowing for the possibility of message drops) and operates over messages and timeouts.

### *7.4 Postmortem log analysis*

Another class of systems is designed for ex post facto analysis of log files. When a bug occurs in production in a distributed system, it is frequently difficult to reproduce or understand: the bug may occur after the system has been executing for days, months, or years, and a the resulting distributed state (on nodes and in the network) can be extremely complex. Developers attempting to understand what happened are faced with megabytes or gigabytes of textual log files. The systems discussed in this section help developers understand such log files. Such tools can be used productively alongside Oddity and other online debugging tools: once a problem has been identified via postmortem tools, developers can further explore the issue—and test attempted fixes—using an online debugger.

Lamport's seminal paper on time and clocks in distributed systems [35] introduced *space-time diagrams* (sometimes called time-space diagrams or Lamport diagrams). Space-time diagrams—influenced by the Minkowski diagrams [44] used in special relativity

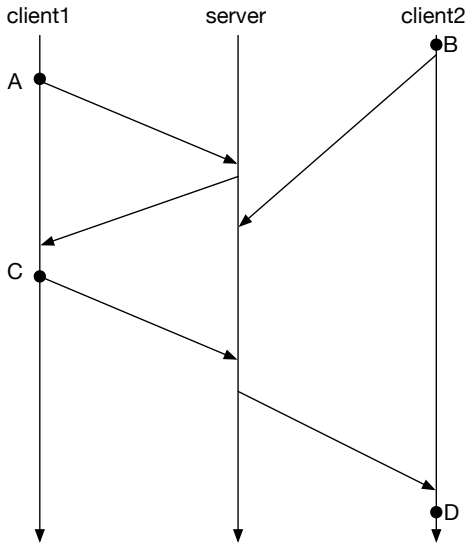


Figure 7.1: A space-time diagram. Each vertical line represents a single node's timeline; diagonal lines between nodes represent messages. Some events at each node are labeled (the letters A, B, C, and D). Since A and C happen at the same node, A *happens-before* C. However, B is unordered relative to both A and C.

to depict the world-lines of observers traveling at different speeds—track the events (message sends, message receipts, and local changes) occurring in a distributed system on two axes: time on the Y axis, and location (machine) on the X axis (sometimes these axes are flipped). Figure 7.1 shows a space-time diagram. Space-time diagrams and similar structures have been used in many visualization systems for system logs [34, 20, 12, 7].

**ShiViz** ShiViz [12], is the current state of the art in space-time diagram-based log visualization systems. The ShiViz log visualization system is built on ShiVector, a Java library that adds vector clocks [23] to a distributed application. ShiVector is responsible for maintaining vector clocks via the algorithm described above; it also appends the current value of the node's vector clock to every line logged by a node (and treats each such

log write as an action, incrementing the node's entry in its vector clock).

The resulting log files (from every node in the application) can then be processed by the ShiViz display engine. ShiViz creates a space-time diagram out of the logs, allowing a developer to see what happened in a given system execution. The diagrams produced by such logs are frequently quite large, so ShiViz includes several features to help developers search and navigate them. A developer can write a parser to extract key-value pairs from log lines; they can then limit the visualization to display only events that include a particular key or value. A developer can also search the log for instances of a particular communication pattern–for instance, request/response pairs between two particular nodes. ShiViz also allows a developer to compare multiple executions, highlighting places where a pair of executions (for example, a normal execution and one where a failure occurred) differ.

Like many similar visualization systems, ShiViz focuses on post-mortem log analysis, so a spacetime diagram showing the whole execution is a natural choice for its visualization. Since Oddity focuses on giving users control over a live system, its user interface focuses on enabling that control. While space-time diagrams present a summary of the entire execution history of a system, Oddity's visualization focuses on allowing the user to understand the current state of the system in detail, while allowing them to navigate to previously-explored states as well.

**Ariadne** Ariande [33] is a postmortmen log visualization tool for both shared-memory and message-passing based concurrent and distributed applications. Unlike ShiViz and similar systems, it uses a tree-based event visualization. In order to use Ariadne, a developer writes a set of hierarchical regular expressions to recognize consecutive patterns of events in a log–for instance, a send and a receipt of a particular message at one layer, and a sequence of expected message transmissions one layer up. Once the developer has described the whole expected execution of the system using these regexes, Ariadne displays a *match tree* on the execution being debugged, with the complete execution at the

root of the tree, user-defined sequences in the middle, and the primitive events emitted by the system at the leaves.

Like ShiViz, Ariadne tracks the happens-before relation between various events in the system. A developer's regular expression specification can include constraints on the relationship between sequences of events: a sequence of events A can *precede*, *parallel*, or *overlap* another sequence of events B if an event in B depends on an event in A but not the other way around, if there are no dependencies, or if each sequence contains a dependency on the other, respectively. Ariadne can thus be used to debug violations of expected control flow between the various nodes in a system.

Ariadne's tree-based visualization represents an alternative to space-time diagrams, focused on presenting a higher-level view of events in a system based on matching sequences of lower-level events. Like ShiViz, it targets post-mortem debugging rather than interactive debugging.

**D3S** D$^3$S [39] (short for "Debugging Deployed Distributed Systems") is a system for monitoring and detecting bugs in deployed, "legacy" (i.e., unmodified) distributed systems. D$^3$S is designed to be used in production, so it has to avoid imposing too large a performance penalty on the running system. In order to use D$^3$S, a developer specifies which data tuples the system should log and a number of predicates D$^3$S should monitor; predicates have access to consistent global snapshots of the tuples produced at every node. The developer specifies the tuples to be included in a snapshot by instructing D$^3$S to instrument specific functions in the node's source code; the tuples are then based on arguments to those functions. D$^3$S instruments each system binary to add the code to expose these tuples to a verifier process, as well as logical clocks (as discussed above) so that consistent global snapshots can be computed. Checking the developer's predicates can itself be distributed using a MapReduce-like [19] model. Once a predicate violation is discovered, the user can examine the sequence of global snapshots that led to the violation. The authors were able to use D3S to find bugs in several systems.

EDL [11], an earlier monitoring system, also logs events and then checks predicates, but (unlike D$^3$S) gives up on correctly handling logical time. Pip [52] was released between EDL and D$^3$S, and checks causal paths in a system against expected behavior.

# Chapter 8

# CONCLUSION

This dissertation has presented Oddity, a graphical interactive debugger for distributed systems. Oddity brings the core features of traditional interactive debuggers—state inspection, fine-grained execution control, and course-grained execution control—to distributed systems developers. Previously, despite decades of use in single-node contexts, interactive debuggers were of limited utility for distributed systems: the behavior of a distributed system is fundamentally determined by the order in which messages and timeouts occur, and traditional debuggers do not allow developers to control this ordering. By simulating a real network and giving users control over its behavior, Oddity allows developers to debug distributed systems. Oddity works on both *executable models*—high-level models of a system's behavior, written in any programming language and implementing a simple event-based API—and, via system call interposition, on arbitrary distributed systems. Regardless of the target system, Oddity supports a novel, general user interface supporting state inspection, execution control, and navigation of multiple alternative execution traces. We have found that students can successfully use Oddity to find and diagnose bugs in their distributed systems.

In the rest of this chapter, we discuss limitations of the current Oddity implementation: areas where Oddity would need to be improved in order for it to be a truly practical debugger for distributed systems. We also discuss several potential avenues for future research work.

## *8.1 Limitations*

The current Oddity implementation has been used successfully in multiple distributed systems classes. It supports state inspection, single-stepping, and watchpoints. It supports both executable models via the Oddity API and many Linux binaries via MajorTom's system call interposition. It does have some limitations, however, that will likely prevent it from being widely used by practitioners in its current form.

**Model checker limitations** In order to automatically search for states matching a given predicate, Oddity includes a model checker. The model checker runs in Oddity's backend, and implements iterative bounded depth-first search in order to find states matching user-supplied predicates. Because of Oddity's architecture, the model checker's performance is somewhat limited—the backend must communicate with the Oddity shim every time an event is executed. Even when Oddity and the shim are being run on the same machine (the likely common case for practical usage) this involves a round-trip through the kernel and a context-switch. This limitation is more or less fundamental—one could implement model checking in Oddity shims themselves, but this would make it significantly harder to generalize to new languages. Another way to improve performance would be to experiment with common model checking optimizations such as dynamic partial order reduction [25] or dynamic interface reduction [26].

**MajorTom limitations** MajorTom currently supports a subset of Linux system calls. It can handle multithreaded systems, UDP and TCP communication, and filesystem usage. It supports annotation of messages, timeouts, and logical state. Since the Linux system call surface is fairly large, supporting all of it would require a massive engineering effort. It is likely, however, that by incrementally adding support for the system calls used by various specific large distributed systems, we could asymptotically approach support for all of the system calls used in practice by systems developers. As discussed in Chapter 6,

we have implemented support for both LogCabin [3] and Redis [5] in MajorTom. With a few more systems, it may be possible to get close to universal distributed system support. Missing subsystems in the current implementation include signal handling (a potential method of interprocess communication) and some TCP options.

## *8.2 Future work*

Oddity consists of multiple interacting components: a graphical user interface, a debugger backend, a model checker, and shims for both executable models and arbitrary systems. Each component can be modified, improved, or replaced independently of the others, providing a modular and extensible platform for research on debugging distributed systems.

### *8.2.1 Interactive space-time diagrams*

In addition to the primary "nodes and inboxes" visualization, Oddity supports visualizing a system's execution as a space-time diagram (e.g., Figure 2.1). Space-time diagrams are useful for viewing a summary of an entire execution trace at once. Because of Oddity's extensible design, adding a traditional (non-interactive) version of space-time diagrams required less than 150 lines of code: Oddity simply pipes a formatted version of the system trace through GraphViz [21], and displays the resulting SVG image in the browser.

In Oddity, space-time diagrams could be enriched with more detailed information about the currently executing system, e.g., by adding JavaScript hooks so that when a user hovers over a node in the space-time SVG, the state of that individual node at that point in history is displayed. Such an enriched space-time diagram would provide a bridge between the "nodes and inboxes" and branching trace history visualizations. Adding these additional features introduces new design and user interaction challenges:

- How should in-flight messages and timeouts be represented?

- How should users interactively control system execution or time travel from such a diagram?

- In what scenarios is one visualization simpler or more effective than another?

Oddity is well-suited for exploring these challenges: the Oddity API abstracts away many of the tedious details for modeling the network, controlling implementations of nodes, and interacting with different programming languages.

### *8.2.2 System-specific interface components*

The Oddity frontend is built around a generic SVG-based canvas which makes integrating other visualization tools straightforward (e.g., for space-time diagrams as discussed above). In particular, Oddity could easily support systems which control aspects of their own visual representation by providing a mechanism to add additional shapes to the frontend SVG. This could be as simple as nodes (optionally) providing a special field in their local state with literal SVG objects to add to the visualization relative to the node's own position. These extensions would enable generic system-specific visualization. For instance, the developer of a state-machine replication system might want to display the log of commands seen at each node as an array of boxes colored by term, while the developer of a ring maintenance system such as Chord might want to display the successor and predecessor of each node as arrows to other nodes. This would involve adding an API for systems to write elements to Oddity's SVG-based interface, and perhaps developing a library of commonly-useful components (such as the arrows mentioned above). In general, Oddity's architecture makes integrating new visualizations easy. Oddity's browser-based frontend also simplifies building on recent advances in data visualization libraries such as D3 [13].

### *8.2.3 Scaling up*

Oddity's current user interface and implementation are well-suited to distributed systems of perhaps a dozen communicating nodes. Past that point, it becomes more useful to reason about and debug the higher-level structure of a distributed system. For instance, a large distributed database such as Google's Spanner [17] shards data across many *clusters* of nodes. Each cluster acts as a single node when communicating with other clusters, but consists internally of many nodes exchanging messages. In order to debug such a system, it would be useful to view the system at multiple different levels of abstraction, either "zooming in" on an individual cluster to see how messages are exchanged internally or "zooming out" to see how the whole system operates, skipping over messages internal to the cluster. Supporting this kind of debugging will require user interface changes as well as backend changes to identify clusters and distinguish between "internal" and "external" messages and timeouts.

### *8.2.4 Other systems challenges*

Handling bugs related to local multithreading (data races, deadlocks, etc.) is an explicit non-goal for Oddity. These bugs remain challenging for developers, however. Adapting Oddity's novel graphical interface to shared-memory multithreading systems running on one node (nodes in Oddity's current interface would correspond to threads) could help developers to diagnose and discover these bugs. This would require graphical representations of synchronization primitives such as mutexes and condition variables, as well as a representation of shared data structures. Such a system could be extended to handle GPU programs, where a crucial issue is the exchange of data between CPU and GPU.

Cryptographic protocols are another potential avenue for graphical step-through debugging research. These protocols involve the exchange of messages between many parties at various levels of trust. In order to understand such a protocol, it is necessary

to understand who can access a given message, who can decrypt it, and which possibilities lead to information leaks. Coming up with good graphical representations of these message attributes could help developers understand and debug high-level descriptions of cryptographic protocols.

### *8.2.5 Industry adoption*

Newcombe et al. [45] have been successful in promoting the use of traditionally academia-bound formal methods tools in the context of distributed systems. These systems' notorious complexity and importance means that business users may be more likely to adopt tools that lead to fewer bugs, even at relatively high cost. As such, we hope to see more practitioners using Oddity or similar interactive debugging tools in the future. Such efforts can be helped by introducing tools in educational settings; since today's students are tomorrow's developers, teaching students to use novel debugging tools is likely to encourage industry adoption of those tools going forward. We have had success deploying Oddity to distributed systems students, and will continue to do so going forward.

# BIBLIOGRAPHY

[1] Akka official website. `http://akka.io/`.

[2] Jepsen official website. `http://jepsen.io/`.

[3] Logcabin official website. `http://logcabin.github.io/`.

[4] Protocol buffers. `http://code.google.com/apis/protocolbuffers/`.

[5] Redis. `http://redis.io/`.

[6] Runway official website. `http://runway.systems/`.

[7] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D. Ernst. Shedding light on distributed system executions. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 598–599. ACM, 2014.

[8] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[9] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 331–346. ACM, 2015.

[10] J. W. Atwood, M. M. Burnett, R. A. Walpole, E. M. Wilcox, and S. Yang. Steering programs via time travel. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 4–11, Sep. 1996.

[11] Peter Bates and Jack C. Wileden. An approach to high-level debugging of distributed systems. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging '83.

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

[13] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12), December 2011.

[14] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[15] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[16] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, November 2009.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, 2012. USENIX Association.

[18] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3), May 2011.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[20] Stephen G. Eick and Amy Wards. An interactive visualization for message sequence charts. WPC '96.

[21] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz open source graph drawing tools. Lecture Notes in Computer Science, pages 483–484, 2001.

[22] Jarret Falkner, Michael Piatek, John P. John, Arvind Krishnamurthy, and Thomas Anderson. Profiling a million user dht. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 129–134, New York, NY, USA, 2007. ACM.

[23] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):5666, 1988.

[24] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[25] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

[26] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 265–278. ACM, 2011.

[27] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[28] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[29] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2), April 2002.

[30] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Profiling services for resource optimization and capacity planning in distributed systems. *Cluster Computing*, 11(4):313–329, Dec 2008.

[31] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Expositor: Scriptable time-travel debugging with first-class traces. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 352–361, Piscataway, NJ, USA, 2013. IEEE Press.

[32] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: language support for building distributed systems. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 179–188. ACM, 2007.

[33] J. Kundu and J. E. Cuny. A scalable, visual interface for debugging with event-based behavioral abstraction. FRONTIERS '99.

[34] Thomas Kunz, David J. Taylor, and James P. Black. Poet: Target-system-independent visualizations of complex distributed executions. *The Computer Journal*, 40(8), 1997.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978.

[36] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

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

[38] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with TLA+. EW 10.

[39] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: debugging deployed distributed systems. NSDI '08.

[40] Philip Maddox. Testing a distributed system. *ACM Queue*, 13(7), July 2015.

[41] Aman Shankar Mathur, Burcu Kulahcioglu Ozkan, and Rupak Majumdar. IDEA: An immersive debugger for actors. Erlang 2018.

[42] Rene McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, 2008.

[43] Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst, and Zachary Tatlock. Teaching rigorous distributed systems with efficient model checking. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 32:1–32:15, New York, NY, USA, 2019. ACM.

[44] Hermann Minkowski. Raum und zeit. *Zeit im Wandel der Zeit*, page 123136, 1988.

[45] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.

[46] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, August 2014.

[47] Diego Ongaro. bug in single-server membership changes, Jun 2015.

[48] Diego Ongaro. Runway: A new tool for distributed systems design. *;login:*, 41(3), 2016.

[49] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. USENIX ATC '14, 2014.

[50] John K. Ousterhout. The role of distributed state. In *In CMU Computer Science: a 25th Anniversary Commemorative*, page pp. ACM Press, 1991.

[51] Patrick Reipschläger, Burcu Kulahcioglu Ozkan, Aman Shankar Mathur, Stefan Gumhold, Rupak Majumdar, and Raimund Dachselt. Debugar: Mixed dimensional displays for immersive debugging of distributed systems. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI EA '18, pages LBW117:1–LBW117:6, New York, NY, USA, 2018. ACM.

[52] Patrick Reynolds, Charles Edwin Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In Larry L. Peterson and Timothy Roscoe, editors, *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings.* USENIX, 2006.

[53] George F. Riley and Thomas R. Henderson. *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[54] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[55] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. NSDI '16.

[56] Andrew P. Tolmach and Andrew W. Appel. Debugging standard ml without reverse engineering. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 1–12, New York, NY, USA, 1990. ACM.

[57] Jeffrey Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 240–250, New York, NY, USA, 2002. ACM.

[58] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. PLDI '15.

[59] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. NSDI '09.

[60] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.