A Graphical Interactive Debugger for Distributed Systems

Doug Woos

March 23, 2018

1 Introduction

Developing correct distributed systems is difficult. Such systems are nondeterministic, since the network can exhibit many different behaviors: behaviors—messages can be dropped or arbitrarily delayed, and nodes can fail and restart. To a large degree, the behavior of the network determines the behavior of a distributed program. What this means in practice is that the "normal" case is much easier to test than the various failure cases: during debugging, it is very likely that messages will be delivered in the order that they are sent and soon after delivery. This means that programmers are unlikely to explore the unusual failure cases in which bugs are likely to hide: for instance, a simplified reconfiguration protocol for the Raft consensus algorithm [3] was discovered to have a bug in the case where two competing reconfiguration requests and a leader failover all interleave. Traditional graphical debuggers, which are designed for code running (perhaps in parallel) on a single machine, are of limited utility in debugging such systems: they do not allow programmers to control which messages will be delivered, and in what order.

DVIZ is a graphical, interactive debugger for distributed systems. It enables engineers to explore and control the execution of their system, including both normal operation and edge cases—message drops, node failures, and delays. DVIZ supports a general execution model: event handlers, written in any programming language, run in response to received messages or timeouts. Event handlers can in turn send messages to other nodes, set timeouts, and modify their local state. DVIZ tracks which messages and timeouts are waiting to be delivered and allows the engineer to control the order. The system can also send state updates to DVIZ so that it can accurately display a summary of each node's current state. DVIZ supports time-travel, allowing engineers 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 of many different cases—for instance, all of the possible orderings of a few messages—in a single debugging session. By enabling programmers to easily explore both normal cases and edge cases—indeed, in DVIZ, no real distinction is made between these cases—DVIZ encourages them to think correctly about distributed systems: rather than assuming the normal case and attaching failure handling as an afterthought, systems must be developed around the possibility of failure and then, if possible, optimize performance for common cases.

DVIZ differs from previous work in several ways. Unlike previous distributed systems visualization systems, it can be used to visualize and control the network behavior of a real system, developed in any programming language. Other systems only visualize the operation

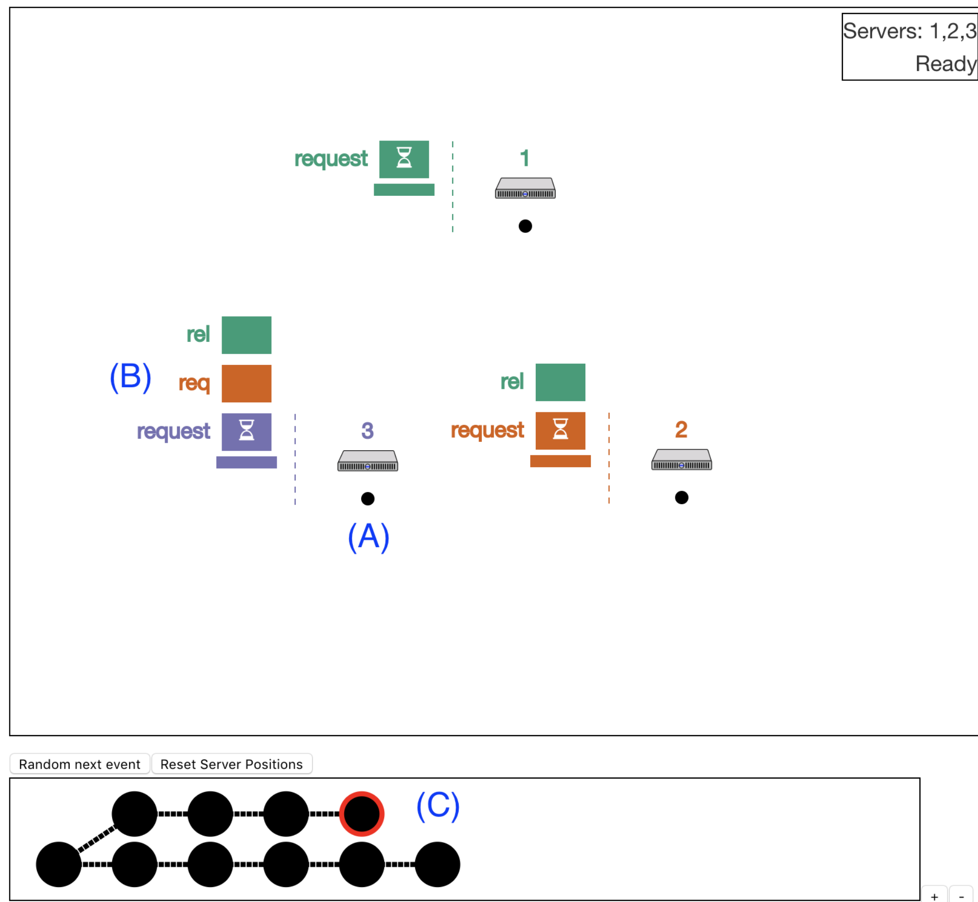

Figure 1: The debugger window. Each node (A) is displayed, along with an inbox (B) of messages and timeouts waiting to be delivered at that node. The user can control delivery by clicking on timeouts and messages, and can also inspect the contents of any message or timeout or the state at any node. Using 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.

of a model, or logs of a particular execution. Similarly, previous debugging systems for distributed systems focused on ex post facto debugging and diagnosis, while DVIZ is geared toward interactive exploration of executions. DVIZ's representation differs from the space-time diagrams used in some previous work in that it is built to allow users to inspect a single state of the system, including in-flight messages and timeouts, in detail while still enabling navigation through an execution (support for traditional space-time diagrams in DVIZ is left for future work).

DVIZ makes the following high-level contributions:

- Interactive debugging of distributed systems. DVIZ is the first system that allows

Table 1: The DVIZ API. In order to use DVIZ, users must 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.

Message	Description	From/To
`register(name)`	Register a node	Node to DVIZ
`start`	Start the node	DVIZ to node
`timeout(type, body)`	Deliver a timeout	DVIZ to node
`message(from, type, body)`	Deliver a message	DVIZ to node
`response(state, messages, timeouts, cleared)`	Response to all events	Node to DVIZ

users to interactively control the order of messages and timeouts that are delivered to each node in a distributed system. DVIZ is designed to encourage and enable users to reason about the correctness of their systems by exploring edge cases as well as normal cases.

- A conceptual model for distributed systems development. In DVIZ, all messages and timeouts for a given node are grouped together in "inboxes," suggesting that any event can occur at any node at any time, and that systems cannot assume a "normal" ordering. DVIZ models history as a tree of possible executions, allowing programmers to navigate multiple executions of their system and to explore the consequences of various event orderings.

- A novel graphical interface. DVIZ includes a new graphical representation (Figure 1) of the partial execution of a distributed system, designed to encourage users to think carefully about the correctness of their systems. This interface allows users to inspect a single state of the system in detail, while also enabling navigation through an execution of the system.

These contributions are expanded below.

2 Contributions

2.1 Interactive debugging of distributed systems

2.1.1 Language-agnostic debugging API

DVIZ provides a simple API, shown in Figure 1, for systems to connect to the DVIZ debugger. This API should be easily implementable in any language, allowing debugging of arbitrary event-based systems. DVIZ currently only supports systems written as deterministic event-handlers, but we do not believe this is a fundamental limitation.

2.1.2 Log visualization

DVIZ can be started from an existing trace of events. This trace could be distilled from production system logs, or produced by a model-checker as a counterexample to some desired

invariant. Visualizing such a trace does not require the system to implement DVIZ's API, but if it does, the user can explore this other alternative executions branching off the initial trace.

2.2 The conceptual model

2.2.1 "Inboxes"

Inboxes contain both the messages and the timeouts waiting to be delivered (in any order) to a node. We believe that this model emphasizes the asynchrony inherent in distributed systems. Other systems model messages as traveling over time between one node and another. In DVIZ, 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. DVIZ'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.

2.2.2 Branching history

The DVIZ debugger models history as a tree of possible executions. Programmers can travel back in time and make different decisions about the network's behavior. This allows users to explore many possible executions of a system, including various interacting failure cases.

2.3 The graphical interface

2.3.1 System state inspection

DVIZ's graphical interface is geared towards representing a single state of the system, including in-flight messages and timeouts, in detail. Users can click to inspect server state or the contents of messages and timeouts. Enabling detailed inspection is crucial for a debugging interface, since users use this information to decide which message or timeout should be delivered next.

2.3.2 Application-agnostic display

DVIZ's display is application-agnostic; it can represent the state and execution history of any distributed system conforming to DVIZ's model (deterministic event handlers). It may be interesting to allow developers to add application-specific interface components (such as a graphical representation of a log for state-machine replication systems); we leave this for future work.

3 Related Work

DVIZ builds on previous work in a few areas: distributed systems correctness, distributed systems log exploration, and distributed systems visualization.

3.1 Correctness

3.1.1 TLA+

TLA+ [1] is a system, based on Lamport's Temporal Logic of Actions, for specifying and reasoning about system models. TLA+ includes a bounded model-checker and a proof system. As described in an Amazon report (Use of Formal Methods at Amazon Web Services), TLA+ is used in industry to model complex protocols and provide evidence that they are correct.

Like TLA+, DVIZ can be used to gain confidence in the correctness of a system model. Unlike TLA+, DVIZ models are executable and can be written in any language.

3.2 Log exploration

3.2.1 D3: Declarative Distributed Debugging

The D3 [2] system allows users to answer debugging queries (such as finding out which sequence of events led to an error occurring) by processing system logs. D3 has a general declarative language for specifying such queries, and is designed to scale to large systems by parallelizing query processing across many nodes.

Like other prior work on debugging distributed systems, D3 is designed for ex post facto debugging of a system running in production, based on log analysis. DVIZ is designed to be used in real time during development.

3.2.2 ShiViz

Like D3, ShiViz is a system for ex post fact exploration of logs generated by production systems. It creates space-time diagrams corresponding to these logs, so that users can visually investigate sources of errors and anomalies. It is not designed for real-time debugging.

3.3 Visualization

3.3.1 Runway

Runway is a system for visualizing models of distributed systems. It consists of a programming language, similar to TLA, 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.

DVIZ's visualization takes some inspiration from by those created in Runway–for instance, as in Runway's Raft visualization, nodes are represented as separate spatial entities laid out in a circle. DVIZ is more general, however, and can be used for any distributed system. Unlike Runway, DVIz does not require engineers to specify their systems in a domain-specific language.

4 Conclusion and future work

DVIZ is the first interactive graphical debugger for distributed systems. IT allows users to control the behavior of the network and observe the execution of the system, and enables debugging multiple executions in order to explore normal- and edge-case behavior.

We have left a number of extensions to DVIZ for future work.

4.1 Space-time diagrams

DVIZ's interface could be extended to include space-time diagrams, which are useful for viewing a summary of a whole execution trace at once. This would only require changes to the graphical interface, but would require some design decisions: how should in-flight messages and timeouts be represented in a space-time diagram? How should users control system execution from the diagram?

4.2 System model

DVIZ currently only supports deterministic event handlers; it could be extended to support nondeterminism, or RPC-based systems with local multithreading. Supporting nondeterminism would require changes to the way DVIZ implements time-travel, since a replay of an execution trace will not necessarily result in the same final state. Supporting RPC-based systems would require a more complex interface between the debugger and the system. Any use of real time by the system would have to go through the debugger, and it may be necessary to represent local background threads as other nodes.

4.3 System-specific interface components

DVIZ could be extended to support system-specific components for visualizing parts of the system state. 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 DVIZ's SVG-based interface, and perhaps developing a library of commonly-useful components (such as the arrows mentioned above).

4.4 Closer model-checker integration

DVIZ can currently be used to animate an execution trace produced as a counterexample by a model-checker, but this integration could be more complete. For example, DVIZ could highlight state components that violate the desired invariant. A model-checker could also be used to provide DVIZ with "breakpoints." Since systems may have to do initial bootstrapping that may be tedious to do manually in DVIZ (for instance, electing an initial leader) the developer could specify that they want to debug the system starting in some state meeting a global property (such as a successful election). DVIZ could ask the model-checker to find such a state, and then allow the user to explore the system's execution starting from the state returned by the model-checker.

References

[1] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and verifying systems with tla+. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 45–48, New York, NY, USA, 2002. ACM.

[2] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: debugging deployed distributed systems. In J. Crowcroft and M. Dahlin, editors, *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 423–437. USENIX Association, 2008.

[3] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319. USENIX Association, 2014.