Visualizing Distributed System Executions

IVAN BESCHASTNIKH, PERRY LIU, ALBERT XING, and PATTY WANG,
University of British Columbia, Canada
YURIY BRUN, University of Massachusetts Amherst, USA
MICHAEL D. ERNST, University of Washington, USA

9

Distributed systems pose unique challenges for software developers. Understanding the system's communication topology and reasoning about concurrent activities of system hosts can be difficult. The standard approach, analyzing system logs, can be a tedious and complex process that involves reconstructing a system log from multiple hosts' logs, reconciling timestamps among hosts with non-synchronized clocks, and understanding what took place during the execution encoded by the log. This article presents a novel approach for tackling three tasks frequently performed during analysis of distributed system executions: (1) understanding the relative ordering of events, (2) searching for specific patterns of interaction between hosts, and (3) identifying structural similarities and differences between pairs of executions. Our approach consists of *XVector*, which instruments distributed systems to capture partial ordering information that encodes the happens-before relation between events, and *ShiViz*, which processes the resulting logs and presents distributed system executions as interactive time-space diagrams. Two user studies with a total of 109 students and a case study with 2 developers showed that our method was effective, helping participants answer statistically significantly more system-comprehension questions correctly, with a very large effect size.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Human-centered computing→ Visualization techniques**;

Additional Key Words and Phrases: Distributed systems, program comprehension, log analysis

ACM Reference format:
Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2020. Visualizing Distributed System Executions. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 9 (March 2020), 38 pages.
https://doi.org/10.1145/3375633

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery grant, funding reference numbers 2014-04870 and 2019-05090. Authors Perry Liu, Albert Xing, and Patty Wang contributed to the project while sponsored by the NSERC USRA program. This material is based upon work supported by the United States Air Force under contract nos. FA8750-12-2-0107 and FA8750-15-C-0010, and by the National Science Foundation under grant nos. CCF-1453474 and CCF-1763423.
Authors' addresses: I. Beschastnikh, P. Liu, A. Xing, and P. Wang, University of British Columbia, 201-2366 Main Mall, Vancouver, BC, V6T 1Z4, Canada; emails: bestchai@cs.ubc.ca, {perry, albert.xing}@alumni.ubc.ca, patty.pcw@gmail.com; Y. Brun, University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA, 01003-9264; email: brun@cs.umass.edu; M. D. Ernst, University of Washington, 185 Stevens Way, Seattle, WA, 98195-2350; email: mernst@cs.washington.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2020/03-ART9 $15.00
https://doi.org/10.1145/3375633

1 INTRODUCTION

Understanding and debugging distributed systems is challenging. Given two events at different hosts, it is not obvious whether one of them is causally dependent on the other, even if each of the events has a timestamp. Distributed systems are prone to failure and are designed to be resilient to it: messages may be dropped, new hosts may join, and existing hosts may leave or fail without notice. Pausing or stepping through a distributed execution is generally impossible. Today, one standard strategy that developers use to diagnose software bugs and reason about program execution is logging [20, 40, 88, 123, 124]. Prior work has observed that logs play an important role in resolving cloud-based system outages [50, 51, 122], access-denied issues [120], and configuration issues [121], among other tasks [85]. A typical way that logging is used in distributed systems is by logging system behavior to generate a log for each host (e.g., using `printf` statements or logging libraries such as Log4J). The developers then analyze the *global* sequence of events across different hosts by serializing the logs from multiple hosts using timestamps in the logs. However, serializing a distributed system to a total order is misleading—even if the clocks are perfectly synchronized, a total order hides the fact that events may have occurred concurrently, independently of one another.

Three important tasks that developers perform while testing or diagnosing distributed systems are:

- ⋆ **Understanding the relative ordering of events.** Given two events at different hosts, is one potentially causally dependent on the other? (If not, then their relative order is an accident of timing—such as scheduling or speed of message delivery—that could be reversed without any effect on the rest of the execution.) When do hosts communicate with one another? What is the minimal set of events that may be responsible for a particular outcome?
- ⋆ **Querying for interaction patterns between hosts.** The distributed system is designed to perform certain patterns of interaction. When do these occur? When do they begin to occur but are interrupted and fail to complete? What patterns exist that may not be documented or may be an emergent property of the system as a whole?
- ⋆ **Identifying structural similarities and differences between pairs of executions.** Given a reference implementation and a buggy implementation, how and when does their behavior differ? Given a single system, characterize the differences between faulty and non-faulty executions. How does the system react to differing environments, such as hardware failure? What are the runtime differences between two algorithms designed to serve the same purpose?

We have developed a novel method for logging and analyzing distributed systems to address these challenges of understanding, querying, and comparing distributed system executions. Our method consists of two parts, *ShiViz* and *XVector*. ShiViz visualizes a distributed system execution as a time-space diagram [66] that explicitly represents the happens-before relation between events (for examples, see Figures 1 and 2). ShiViz contains features to support the following system understanding tasks:

- ⋆ **Event ordering.** The ShiViz time-space diagram explicitly but compactly represents the relative ordering of events across hosts in the system, capturing concurrency among events. The visualization also links the time-space diagram to the corresponding textual entries in the log. ShiViz enables the developer to simplify the graph by transforming it to elide information that is not relevant to their current task.
- ⋆ **Interaction patterns.** ShiViz allows developers to construct event sub-graphs and search for them in the time-space diagram. These sub-graphs may include constraints on host identifiers and event meta-data.

- ⋆ **Multiple execution comparison.** ShiViz can present two execution graphs side-by-side to help developers compare these executions. ShiViz includes algorithms to highlight differences between pairs of executions and supports the clustering of executions based on features.

ShiViz requires the distributed system to generate a particular kind of log, which captures concurrency information. XVector, a suite of libraries for C, C++, Java, and Go, helps automate this process.[1] XVector interposes on communication and logging channels at each host in the system to add vector clock timestamps [36, 80]. These vector timestamps capture the *happens-before* relation [68]. XVector produces a textual log of printf-style messages augmented with vector clock timestamps. ShiViz reads these logs to reconstruct the graph of inter-host communication and display a time-space diagram. ShiViz includes specific capabilities to help developers implement correct systems. The capabilities help a developer understand event ordering, query for interaction patterns, and compare executions.

We evaluated the behavior-understanding capabilities in ShiViz through three studies:

(1) We ran a controlled experiment with a mix of 39 undergraduate and graduate students. One group of participants studied distributed system executions using ShiViz and another group without ShiViz. The study asked all participants to answer questions about the system represented by the executions.
(2) 70 students in a distributed systems course used ShiViz as part of two homework assignments to help them debug and understand their implementations.
(3) We ran a case study with two systems researchers who were developing complex distributed systems to evaluate the end-to-end usefulness of ShiViz to developers in their work. Across these studies, we collected the developers' impressions via surveys and interviews.

Our evaluation results demonstrate that ShiViz supports both novice and advanced developers in distributed system development tasks. For example, our controlled experiment with 39 participants showed that those using ShiViz answered statistically significantly more distributed system understanding questions correctly than control-group participants without ShiViz, with a very large effect size. The two case studies provide qualitative data about the ShiViz developer experience, indicating that ShiViz helped these participants solve their problems faster than if they were to use other tools.

This article's main research contributions are:

- ⋆ A new method (and the supporting open-source implementation for systems written in C, C++, Java, and Go, as detailed below) for logging and analyzing distributed systems to support common system-understanding developer tasks.
- ⋆ Advanced, composable graph transformations for manipulating distributed system execution graphs, including constrained custom structured search and filtering by process.
- ⋆ A mechanism for side-by-side juxtaposition of pairs of execution graphs, supporting all of the single-graph transformations, as well as new transformations to highlight graph differences and similarities.
- ⋆ Algorithms to cluster distributed executions using two approaches computed over sets of graphs: clustering by similarity to a specified graph and by number of processes.

[1]For simplicity, we refer to any one of the libraries as *XVector*.

We also contribute two open-source, publicly available implementations:

- ★ ShiViz, a robust, web-deployed, freely available implementation of the above contributions for developers to use for distributed system development. ShiViz is available online: http://bestchai.bitbucket.io/shiviz/.

  A video demonstrating key features of ShiViz is also available: http://bestchai.bitbucket.io/shiviz-demo/.

  ShiViz is being actively used in the research community [82, 107] and by projects within companies like Microsoft[2] and by popular open source projects like Akka.[3]
- ★ Four implementations of XVector, easy-to-use libraries for logging distributed system executions, for C, C++, Java, and Go. These libraries are available online: https://github.com/DistributedClocks.

The rest of this article is structured as follows: Section 2 illustrates distributed systems challenges with three use cases. Section 3 presents distributed systems background. Section 4 presents ShiViz's mechanisms for navigating and manipulating the visualization, which address the challenges of distributed system understanding. Section 5 describes our XVector and ShiViz implementations. Section 6 evaluates XVector and ShiViz with a series of controlled experiments and user studies. Section 7 discusses the threats to the validity of our study. Section 8 describes the work's limitations and future work. Section 9 places our work in the context of related research. Section 10 summarizes our contributions.

2 EXAMPLE USE CASES

This section presents three use cases that highlight how ShiViz supports developers in debugging distributed systems. These use cases were inspired by our own experiences in building distributed systems, as well as by conversations with real distributed system developers.

1. A developer attempts to understand why two leader hosts are active simultaneously in her implementation of a leader election algorithm.

A safety invariant in a leader election algorithm is that at any given time, there is at most one leader in the system. If a leader fails (such as crashing or becoming disconnected from the network), then a new leader is elected.

Consider a developer whose implementation sometimes generates a situation in which two leaders are active simultaneously. The developer wants to understand under what circumstances this situation occurs.

It is standard practice for the developer to add print statements to capture critical state transitions of the system. For example, when a host becomes the leader, the host logs a message to this effect. To use ShiViz, the developer takes three additional actions. (1) The developer instruments the system using the XVector library, which augments each logged message generated by the system with a vector timestamp. (2) The developer deploys the instrumented system and reproduces the problem. The system writes logs augmented with vector timestamps. (3) The developer runs ShiViz to visualize the logs from all of the hosts.

The developer first wants to know where in the trace the system entered the invalid two-leader state. She finds events where a host became a leader via a **keyword search** for `isLeader=true && desc='became leader'` (these keywords are specific to the system logging the developer has implemented). The developer observes that for this system execution, there were four matching

[2] https://github.com/p-org/TraceVisualizer.
[3] https://github.com/akka/akka-logging-convertor.

results. She navigates to each of the matched messages, one at a time, and finds that two of the results have the same "epoch" field value, which violates a correctness property of the algorithm.

Next, the developer considers the context **surrounding** the two contradictory messages. By studying the relative ordering of events around these two messages, she sees that one of the hosts, prior to becoming a leader, knew of the other host as the leader. She also sees that the messages preceding this host's change of state to being a leader are "send" messages to the leader. These messages, however, never receive a reply from the current leader. This eventually causes the local host to declare itself the leader without incrementing the epoch number.

2. A developer wants to know why a replicated data storage sometimes fails to reply to client requests.

In this system, a client contacts the front-end host, which delegates the client to one of the replicated data storage hosts at random. Some clients that contact a replicated data storage system time out waiting for a reply. However, this happens rarely and non-deterministically.

The developer begins by instrumenting the system with XVector to log vector timestamps, then runs the system and uses ShiViz to visualize the log of external (client interactions) and internal events.

The developer formulates a **structured search query** to find cases in the recorded execution where a client sent a request but did not receive a response. To do this, the developer uses ShiViz's graphical interface to describe the scenario visually, as a sub-graph, shown in Figure 5 in Section 4.2. In the query sub-graph, a message from a client host reaches the front-end in the system, the request is forwarded to some data storage host, but the client does not receive a reply from the front-end within three logged events.

The search returns five locations in the execution graph where the scenario occurred. Looking over the five locations that match this scenario, the developer sees a pattern: The requests that do not receive a response are forwarded to data storage hosts that have been recently added to the storage system. This indicates that a firewall may be the root cause, preventing the storage hosts from communicating back to the front-end because the firewall configuration has not been updated to allow traffic from these hosts to reach the front-end server.

3. A developer is implementing a client library for the SMTP mail transfer protocol. During testing he learns that his implementation does not interoperate with a standard SMTP server. He wants to know how his SMTP implementation differs from other clients.

In implementing his SMTP library, the developer follows the original RFC specification [61]. In response to a test message, an SMTP server responds with `503 Bad sequence of commands` and closes the connection.

The developer decides to compare his client-side implementation of the SMTP protocol to a different client-side implementation, Mozilla Thunderbird, which he knows works properly with the same SMTP server. He instruments both SMTP clients and the SMTP server with XVector. He generates an execution trace containing logs for Thunderbird and the server, and then generates an execution trace containing logs for his library and the server. He loads the two executions into ShiViz for analysis.

The developer uses the **side-by-side execution comparison** ShiViz feature to compare the two executions. Then, he uses the **highlight differences** ShiViz feature to highlight those events that appear in one execution but do not appear in the other execution. This view reveals that his library never sends the required `RCPT` command to the server.

By contrast, to resolve the issues in these three use cases *without* ShiViz, a developer today would (1) add logging code, such as print statements, automatically or manually to the system, and (2) study the resulting *textual* logs, one per node in the system. As a result, without ShiViz, the developer cannot easily understand the partial ordering of concurrent events in the system, and has to piece together what happened by scanning through the individual node logs. Section 9 considers specific tool alternatives and prior research work in more detail.

Summary. These three use cases illustrate the utility of ShiViz's mechanisms for (1) understanding the relative ordering of events in an execution, (2) querying for patterns of interaction between hosts in an execution, and (3) identifying structural similarities and differences between pairs of executions.

3 BACKGROUND: DISTRIBUTED SYSTEMS

This section overviews distributed systems concepts necessary to understand our work.

3.1 Definitions

A distributed system is composed of a number of hosts, and each host generates a totally ordered sequence of *events*. Each event is associated with a set of attributes, such as the time the event occurred. A *host trace* is the set of all events generated at one host.

A *system trace* is the union of a set of host traces (one per host) corresponding to a single execution of the system. A *log* is a set of system traces. A log represents multiple executions.

Order is an important property of a distributed execution. Events are ordered in two ways. First, the *host ordering* orders every pair of events at the same host, but does not order events that occur at different hosts. Second, the *interaction ordering* orders dependent events at different hosts; for example, if two hosts use message passing to communicate, a *send message* event is ordered before the *receive message* event for the same message. Taken together, the host and interaction ordering generate a partial ordering over all events in the execution. The partial ordering is known as the *happens-before* relation [68]. Understanding this partial order is central to most tasks that a distributed system developer performs.

Figure 1 is a *time-space diagram*, which is the standard visualization of the happens-before relation in a distributed system [66]. (ShiViz displays distributed system executions as time-space diagrams; see Figure 2.) The time-space diagram expresses happens-before relations as directed edges between events. (This ordering is transitive, but for clarity, the diagram omits transitive edges.) For example, in the diagram, the *tx prepare* event at TX Manager occurs before the *abort* event at Replica 1 (there is an edge from *tx prepare* to *abort*). In a time-space diagram, two events are concurrent if the time-space diagram lacks a directed path between them. For example, in Figure 1 there is no directed path between the concurrent events *abort* at Replica 1 and *commit* at Replica 2.

One way to represent partial order in a system trace is to associate a *vector clock timestamp* with each event. These timestamps make explicit the partial order of events in the system trace. The remainder of this section briefly explains vector time and an algorithm to record vector timestamps in a distributed system [36, 80]. Though more efficient vector clock mechanisms exist [4, 6], we believe that vector timestamps [36, 80] are practical for debugging: short time periods on large systems, or during development and testing.

For concreteness, our explanation below assumes that the distributed system uses message passing, but vector timestamps are equally applicable to a system that uses other mechanisms for inter-host communication, such as shared memory.

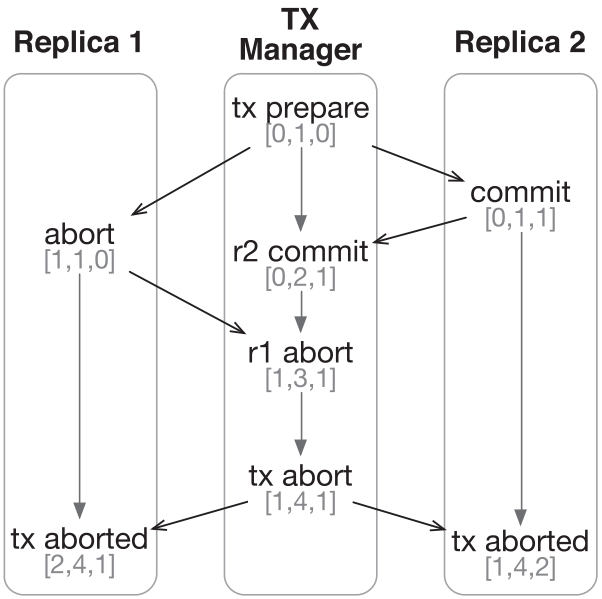

Fig. 1. Time-space diagram illustrating an execution of the two-phase commit protocol [8] with one transaction manager and two replicas. The vertical lines represent the host ordering, and the diagonal lines represent the interaction ordering. Each event has an associated vector timestamp. For conciseness, this time-space diagram does not explicitly include message send and receive events.

3.2 Ordering Events with Vector Time

In a distributed system of h hosts, each host maintains a local logical time. In addition, each host maintains a vector clock, which is an underestimate of the logical time at all hosts. A vector clock is an array of monotonically increasing clocks $C = [c_0, c_1, \ldots, c_{h-1}]$, where c_j represents the local logical time at host j. C_i denotes the vector clock at host i and $C_i[j]$ represents i's current knowledge about the local logical time at j.

The hosts update their clocks to reflect the execution of events in the system by following three rules:

(1) Each host starts with an initial vector clock $[0, \ldots, 0]$.
(2) After a host i generates an event, it increments its own clock value (at index i) by 1, i.e., $C_i[i]$ ++. Sending and receiving a message are each considered an event.
(3) Every message sent by a host i includes the value of its vector clock C_i. Upon receiving the message, host j updates its vector clock to C'_j such that $\forall k, C'_j[k] = \max(C_i[k], C_j[k])$.

The above description assumes that every host knows the complete set of participating hosts in the system and that this set does not change over time.

Using the above procedure, each event in the system is associated with a *vector timestamp*—the value of C immediately after the event occurred. Vector timestamps are partially ordered by the relation $\prec$, where $C \prec C'$ if and only if each entry of C is less than or equal to the corresponding entry of C' and at least one entry is strictly less. More formally: $C \prec C'$ iff $\forall i, C[i] \leq C'[i]$ and $\exists j, C[j] < C'[j]$. This ordering is partial, because some timestamp pairs cannot be ordered (e.g., $[1, 2]$ and $[2, 1]$). In this case, we say that the two corresponding events occurred *concurrently*.

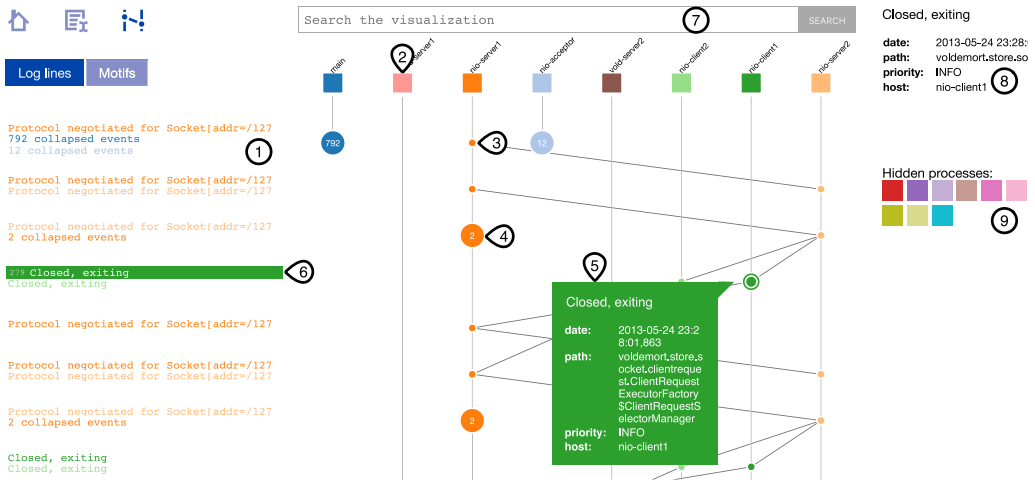

Fig. 2. ShiViz screen visualizing an execution of the Voldemort distributed data store. Key features and areas of the UI are numbered. ① Log lines corresponding to the currently visible time-space diagram to the right. ② Boxes at the top represent hosts in the system; the box colors provide a consistent coloring for events and log lines associated with a host. ③ An event on a host timeline is represented as a circle. ④ A sequence of local events with no intermediate communication is collapsed into a larger circle whose label indicates the number of collapsed events. ⑤ When an event is clicked, details for the event are shown in a pane, and ⑥ the relevant log line is highlighted and its log line number is shown to its left. The log line may also be clicked directly. ⑦ The graph may be searched by keywords or by structure (Section 4.2). ⑧ When the developer hovers over an event or host, details are shown in this top-right pane. ⑨ The developer can click on a host to omit it and its log lines from the visualization (Section 4.1.3). Hidden hosts are kept in a list to the right and can be restored with a double-click.

Note that it is possible for one event to precede another event in wall clock time, yet the two events are concurrent according to their vector timestamps.

4 SUPPORT FOR LOG UNDERSTANDING

ShiViz visualizes a logged distributed execution as a time-space diagram—a common mental model used by programmers for distributed systems. Two key differences from prior tools are ShiViz's emphasis on time-space visualizations that it can generate *for all* concurrent systems and a powerful set of operations to help developers navigate, search, and explore the logged execution as a graph rather than a text-based log.

The following three sections explain how ShiViz supports developers in the three key tasks that motivate our work: understanding the relative ordering of events in an execution (Section 4.1), querying for patterns of interaction between hosts in an execution (Section 4.2), and identifying structural similarities and differences between pairs of executions (Section 4.3). The reader may use the deployed version of ShiViz while reading this section at http://bestchai.bitbucket.io/shiviz/.

4.1 Exposing the Global Event Ordering

A key challenge in understanding a distributed system is the interleaving of concurrent events. A log-based view of an execution cannot clearly convey the inherent partial ordering in such a system.

ShiViz displays two types of information: the log itself and the time-space diagram derived from it. By presenting the views simultaneously, ShiViz provides the developer with the familiar

log-based view, as well as a view that conveys the partial ordering of what happened. Figure 2 shows a screenshot of the main ShiViz screen. This screen displays the logged execution in two ways: textually, as log lines, on the left, and graphically, as a time-space diagram, on the right.

In the time-space diagram, time flows from top to bottom. The boxes at the top represent hosts and the vertical lines below them are the host timelines. Circles on a host timeline represent events executed by that host. Edges connect events, representing the recorded happens-before relation: an event that is higher in the graph happened before an event positioned lower in the graph that it is connected to. The hosts are ordered from left to right in decreasing number of events executed; the developer may change this ordering.[4]

To reveal more log lines and events in the execution, the developer scrolls down. During scrolling, the host boxes remain in their positions at the top of the window to provide context.

The goal of the time-space diagram is to reveal ordering between events at different hosts, including likely chains of causality between such events. Such information is not easily discernible from manual inspection of logs. The execution graph reproduces the events and inter-host communication captured in the input log, but the graph makes the ordering information explicit in the visualization. For example, the developer in use case 1 in Section 2 wants to understand the context **surrounding** two contradictory messages. ShiViz exposes this context, including the ordering of events, the events themselves, as well as the corresponding log lines.

4.1.1 Relating the Time-space Diagram with Log Lines. Two views (log-based and the partial ordering) of the same execution can create confusion, since many tasks require using both types of information—what was logged and when it was logged. To help with this, ShiViz augments, not replaces, the log with an execution graph—the input log is always accessible in the left panel and is *linked* to the execution graph through visual cues. Hovering over an event or clicking on an event in the graph highlights the background of the corresponding log line in the left panel. This informative feedback supports the developer's mental model that the execution graph is a more structured representation of the log. For example, the selected event ⑤ in Figure 2 corresponds to the highlighted log line ⑥ `Closed, exiting`. Likewise, clicking on a log line on the left highlights the corresponding event in the graph.

The log lines are positioned to horizontally align with their corresponding events to strengthen the association. However, this also means that the log lines may appear in an order different from the order of log lines in the input log, but always one consistent with the partial order and one that could have been generated by the same execution.

4.1.2 Associating Log Lines with Hosts. To make it easy to spot the lines in the log associated with a specific host, each host in the execution graph is associated with a color that is used for its host box, event circles, and log lines. This color-coding works best when displaying few enough hosts to allow ShiViz not to reuse host colors.

4.1.3 Graph Transformations. The execution graph may contain a large number of events, including ones that are not relevant to a developer's task. ShiViz includes three graph transformations, shown in Figure 3, to help developers focus on relevant behavior in the execution.

Transformation 1: collapse local events. ShiViz emphasizes ordering information and communication between processes. Series of local events that do not involve interactions with other processes are often less relevant to understanding the communication patterns in the system.

[4]ShiViz allows users to order the hosts in one of four ways: (1) Order hosts in descending order of the number of events each host generated in total. (2) Order hosts in ascending order of the number of events each host generated in total. (3) Order hosts in descending order of the line number where the host appeared in the log. (4) Order hosts in ascending order of the line number where the host appeared in the log.

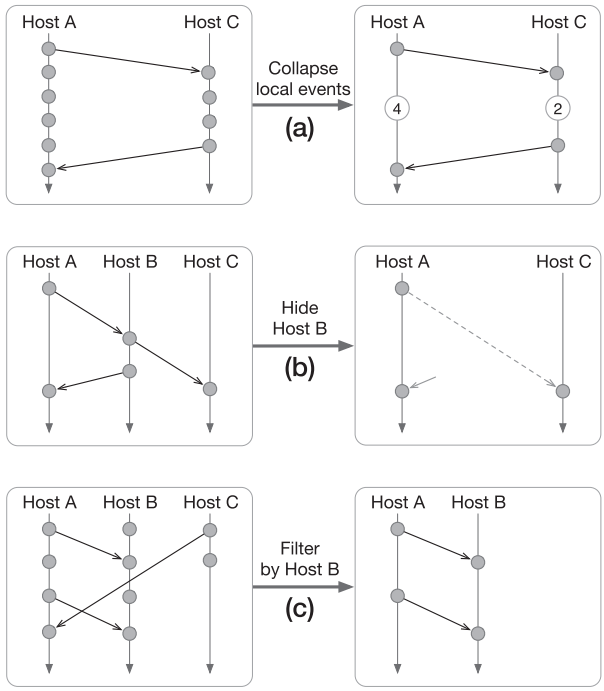

Fig. 3. **(a)** A collapse transformation collapses a series of local events on a host timeline. **(b)** A hide host timeline transformation removes a host timeline from view. **(c)** A filter by host transformation retains just those hosts and events that are relevant to communication with the specified host(s).

ShiViz groups and merges these local events into a single node to simplify the visualization and emphasize global ordering information, as illustrated in Figure 3(a). This transformation is enabled by default: All series of events that can be collapsed are collapsed when the execution graph is first shown to the developer. They can expand or collapse each set of nodes independently by clicking on the node and selecting "expand" or "collapse."

Transformation 2: hide host timeline. Frequently, a developer is interested in the behavior at a subset of the hosts in the system. Using the hide host transformations, a developer can remove a host (including the corresponding host timeline and log lines) from view, or bring these back into view. To hide a host, a developer double-clicks on the host box at the top of the time-space diagram. Hidden hosts are collected in the right panel (⑨ in Figure 2). To unhide the host, developers double click on the host box in this panel. A dashed edge connects two events at not-hidden hosts that are connected transitively through one or more hidden hosts. Figure 3(b) illustrates this transformation.

Transformation 3: filter by host. The filter by host transformation refines the graph to show just the set of events and hosts that are relevant to communication with a particular host. A white square inside the host box denotes a filtered host. It is possible to filter by more than one host and to unfilter hosts in any order. Figure 3(c) illustrates the filter by host transformation. Figure 4 shows an example of this transformation on the log and view from Figure 2.

Throughout these transformations, the log in the left panel continues to reflect the *visible* events on the right. For example, when a host is hidden from the graph, the log lines for that host are hidden as well. Therefore, these three graph transformations are log transformations as well.

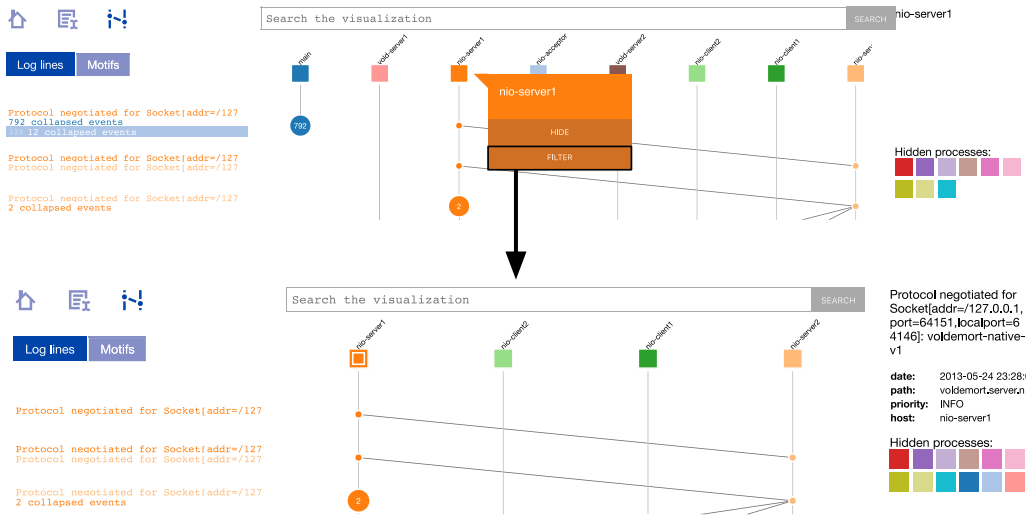

Fig. 4. An illustration of the filter by host transformation, applied to the log and view in Figure 2. (Top) The user can select to hide a host or to filter by a host transformation by clicking on the host's box. (Bottom) Shows the result of choosing to filter by a host (nio-server1) in the top view. Notice that the filtered orange host box is highlighted and the result of the transformation is to retain just those host timelines with which this host has directly communicated. The boxes for hosts with which the filtered host did not communicate are added to the hidden processes list on the right and their timelines are removed from view.

A key feature of ShiViz transformations is that they are composable; that is, ShiViz has built-in semantics to resolve the layering of transformations. So, it is possible to hide a host, then uncollapse a set of events, filter by two hosts, and finally hide one more host. A developer can also remove transformations from this layering in any order, not just in the reverse of the order in which the transformations were added.

4.2 Supporting Execution Graph Queries

Few developers read or browse a complex system's log manually. Instead, developers often focus on specific parts of the log and write regular expressions to parse information they need out of the log [109]. Presenting a visual partial ordering that describes an execution cannot help with this. To be useful for complex logs, ShiViz must be able to support developer queries about the partial ordering. A key challenge for ShiViz is in supporting *graph-based* queries. Using graph-based queries, developers can express the topologies of interesting event orderings, rather than the text contents of the events themselves.

ShiViz implements two kinds of searches: structured search and keyword search. When a developer clicks in the search area at the top of the screen (⑦ in Figure 2), a drop-down appears, using which the developer can either perform a structured search or a keyword search query.

4.2.1 Structured Search. In a structured search, a developer queries the execution graph for a set of events that are related through a particular ordering pattern. We call this pattern a *sub-graph*. The sub-graph can be either user-defined or predefined, and search results are presented as highlighted sub-graphs overlaid on top of the execution graph.

A user-defined sub-graph (left side of Figure 5) is generated by the developer with a mouse by adding a number of hosts, host events, and connections between pairs of events to define a partial order. During a query, ShiViz searches every possible permutation of hosts and events to

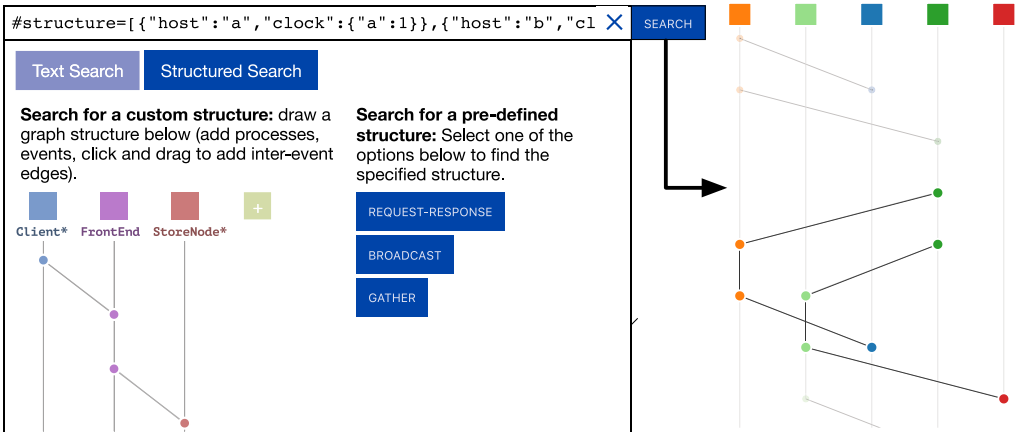

Fig. 5. Structured search with a query corresponding to use case 2 from Section 2. The regular expressions (in **bold** font below each host box) are host constraints that restrict the search to those scenarios in which the host names in the execution graph match the three host constraints. The right side shows the results of the search with two highlighted matches.

find matching sub-graphs. Most interesting user-defined sub-graphs are small in size (fewer than 5 hosts and 10 events) and our algorithm is fast enough to find a match, if one exists, in under a second.

In Figure 5, the sub-graph represents a forwarding pattern where the leftmost host sends a message to the middle host, which then communicates with the right-most host. This communication chain is then reversed. The drawn sub-graph is also represented as a textual query in the search bar in Figure 5, specifying hosts and their corresponding vector clocks. Modifying either the user-defined sub-graph or its textual representation will update the other to match. This simplifies custom structure reuse and sharing, as sub-graphs can be generated by pasting text strings starting with `#structure` into the search bar.

When searching for a user-defined sub-graph, ShiViz queries the execution graph for an exact match of the custom structure; that is, only events connected in the specified order, without any interleaving events, will be shown as a search result. Developers can further filter the result set by specifying host-name constraints. A host timeline with an added constraint will only map to timelines in the execution graph that satisfy the given restriction. For example, the right side of Figure 5 shows how ShiViz visualizes two highlighted matching patterns corresponding to the custom query on the left.

The query in Figure 5 corresponds to use case 2 in Section 2; it is an example of a user-defined sub-graph with host-name constraints. In this figure, the `Client*` constraint on the left-most host constrains the structured search to match host names like `Client-1`.

Alternatively, developers can search for a predefined sub-graph pattern (middle of Figure 5). ShiViz has three such patterns: (1) request-response: a source host sends a request and the destination host sends a response back; (2) broadcast: a host sends a message to most other hosts in the system; (3) gather: a host receives a message from most other hosts. Figure 6 lists example search results for these three predefined sub-graph patterns. The predefined searches are more advanced than the custom structured search and can find variations of an underlying pattern. For instance, a broadcast can be represented as a sequence of separate send events, as illustrated in blue in Figure 6, *or* as multiple sends at once, as seen in green in the same figure. Both structures will be highlighted as a search result. The predefined sub-graphs are also more flexible and

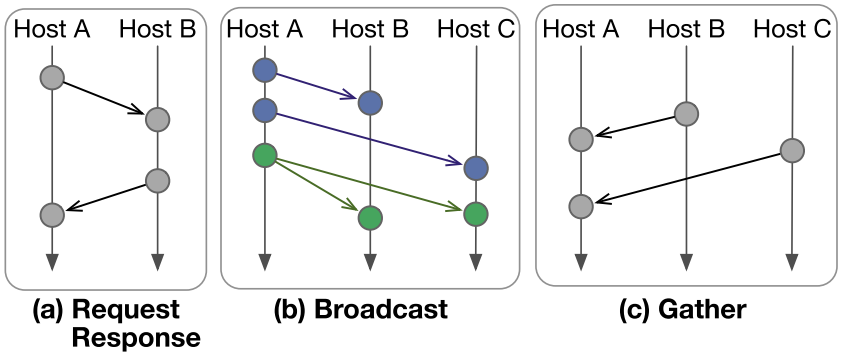

Fig. 6. Example search results for three predefined sub-graph search patterns in ShiViz.

not restricted to exact matches. For instance, when querying for request-response interactions, events interleaving between the request and response events will not interfere with detection of the pattern.

Both the user-defined and predefined searches allow developers to locate communication patterns of interest within an execution graph. The presence or absence of queried sub-graphs at particular points in an execution can help developers detect anomalous behavior, aiding them in their debugging efforts.

4.2.2 Keyword Search. Unlike the structured search, which allows the developer to find patterns in the structure of the graph, a keyword search allows a developer to search across the events' data fields.

Developers can query for a particular word or phrase and find all events in the graph with a field matching the search. These constraints can be generalized with JavaScript regular expressions and combined through logical connectives.

In both types of searches, the developer interacts with the search results in one of two ways. The developer can scroll through the execution graph to find the highlighted search results manually, or they can jump to the previous/next instance in the results using a pair of arrow buttons in the search bar.

4.3 Multi-execution Comparing and Clustering

A common challenge for studying a concurrent system over several executions is the nondeterminism that makes two executions not directly comparable. That is, an interleaving difference may make two logs appear behaviorally different even though the system that produced the logs is the same. For this reason, we decided to extend ShiViz to support developers in understanding and juxtaposing multiple executions recorded in the log. Specifically, when ShiViz parses more than one execution from the log, it allows the developer to view any single execution or any pair of executions.

The features described previously (e.g., search, hiding a host) continue to operate in the side-by-side view. For example, if the developer hides a host in the left execution, the same host will be hidden in the right execution.

4.3.1 Highlighting Differences between Executions. A feature exclusive to the pairwise view is the ability to highlight differences between two executions. To use this feature the user selects a *base execution*, against which other executions will be compared. Hosts or events in an execution that do not appear in the base execution are drawn as rhombuses (Figure 7). This highlighting is

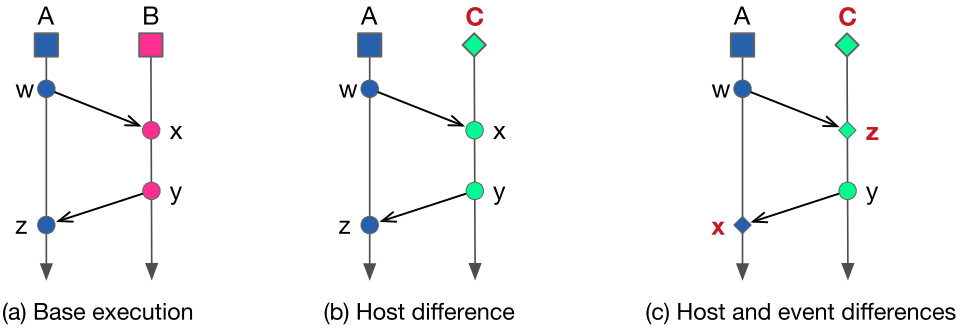

Fig. 7. ShiViz uses rhombuses to highlight differences between executions. Here, compared to a base execution (a), the execution in (b) has a different host, and the execution in (c) differs in both hosts and events.

done by walking the timeline of corresponding processes and comparing each event against events in the base execution.

This explicit highlighting of differences provides developers with fast detection of anomalous events, or points where two executions diverge, allowing them to more effectively compare executions. For example, during a debugging task, this differencing can lead the developer to quickly spot buggy behavior.

The event differencing mechanism compares the host name and event text strings. As future work, we plan to allow the developer to specify which other data fields to include in event comparison and to highlight order or structural differences between two executions.

4.3.2 Clustering Executions. ShiViz also supports grouping multiple executions into clusters. Developers can cluster by the number of processes or by using the comparison against a base execution described above (Figure 7). Cluster results are presented as distinct groups of listed execution names.

Execution clusters aid in the inspection and comparison of multiple executions by providing an overview of all executions at once. Developers can quickly scan through cluster results to see how executions are alike or different based on the groups into which they are sorted. Clustering also helps developers pinpoint executions of interest by allowing them to inspect a subset of executions matching a desired measure. This subset can be further narrowed down by performing one of the previously mentioned searches (keyword and structured search) on top of the clustering results. Execution names among clusters are highlighted if their corresponding graphs contain instances matching the developer query.

Section 6.2.2 discusses two case studies in which the multi-execution comparison and clustering were useful to developers working on complex distributed systems.

5 IMPLEMENTATION

ShiViz reads events and their happens-before relation from a log, then visualizes one or more executions as time-space diagrams. Sections 5.1 and 5.2 describe creation and parsing of logs, and Section 5.3 gives details about the ShiViz implementation.

5.1 Logging the Happens-before Relation

Typical distributed systems already contain logging calls to record events and values of interest; for example, by printing them to a file. Therefore, most of the work to create a log is already done. However, the partial happens-before order of the events may be lost due to the concurrent execution. One way to preserve this order is to associate a *vector clock timestamp* with each

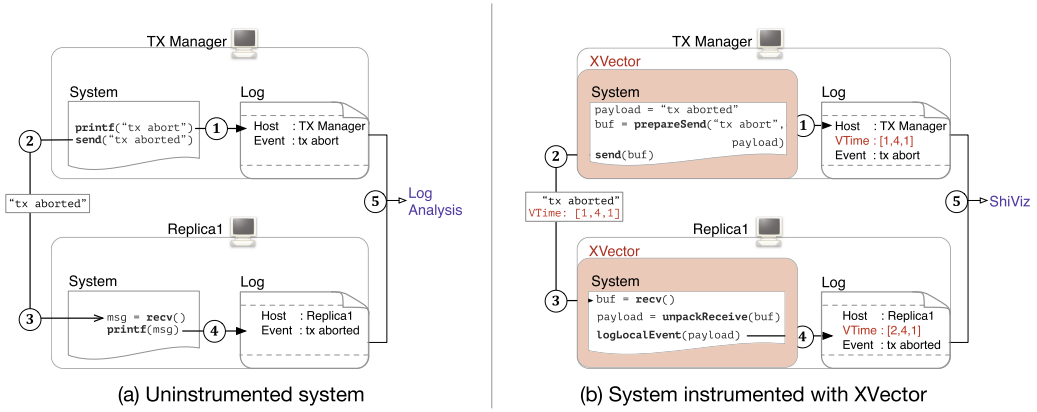

Fig. 8. A log-based view of the bottom-most edge from TX Manager to Replica 1 in Figure 1. (a) Uninstrumented version of the system that produces a log that can be manually analyzed. (b) A system instrumented with XVector to capture the partial ordering relation; the resulting log can be analyzed using ShiViz.

event [36, 80]. We have developed XVector, a suite of libraries to instrument distributed systems to log happens-before partial order via vector time. For exposition, our XVector explanation assumes the distributed system uses message passing, but the XVector approach is equally applicable to systems that use other inter-host communication, such as shared memory.

The XVector library relieves developers from needing to implement the vector clock algorithm [36, 80] (recall Section 3.2). To use the library, developers (1) initialize the library, and (2) make calls to the library to pack and unpack existing messages sent by the system with a header that consists of the vector clock timestamp. XVector maintains a vector clock for each host in the system and updates it whenever the client application makes calls to pack or unpack. These packing calls also take a string that will be logged to a log file along with the current vector clock timestamp. XVector also includes a call that can be used to just log a message with a vector timestamp. Figure 8 details the XVector instrumentation for a small part of the two-phase commit execution whose time-space diagram is pictured in Figure 1.

We have built XVector implementations for several languages: C, C++, Java, and Go. These libraries inter-operate, allowing multi-lingual systems to be instrumented. XVector integrates with popular logging libraries in these languages, such as Log4J. For two languages, Java and Go, we have versions of XVector that automatically instrument the source code using static analysis and a whitelist of known networking library calls. This procedure is described in more detail in our previous work on inferring distributed system invariants [46]. For the other languages, the developer must manually log information with XVector. We are working on automating this process for all languages. The XVector libraries are open-source: https://github.com/DistributedClocks.

5.2 Log Parsing

Given a log, ShiViz needs to know the following:

- What parts of the log represent events.
  - —For each event, the host that executed the event, the vector timestamp of the event (recall Section 5.1), and a text string describing the event.
  - —Optionally, additional data associated with each event, which developers can search for within ShiViz (see Section 4.2).
- Optionally, how to divide the log into multiple system execution traces.

The developer specifies this information with a regular expression that matches events and an optional regular expression to delimit system traces. The event regular expression must contain three named capture groups to extract the triples of `[host,clock,event]` information from the log and may contain other named capture groups.

This method of processing logs is flexible and can accommodate many format types (including text, JSON, etc.) without making any assumptions about the system that produced the log. We have used ShiViz to visualize not just distributed systems but also multi-threaded applications.

5.3 ShiViz Implementation

After processing the log, ShiViz presents the developer with a view like the one in Figure 2, which is a screenshot of ShiViz for an input log from a distributed data store system called Voldemort [112].

ShiViz is implemented as a pure client-side web application, making it trivial to deploy. To use it, a developer only needs a web browser. A developer may either paste logs and regular expressions into the web form, or they can upload a local file containing the logged execution(s) and necessary regular expression(s). ShiViz implements all of the logic in JavaScript, making it highly portable.

ShiViz never sends the input system logs over the network. This makes it safe to use in a corporate environment where logs may contain sensitive information.

ShiViz is an open source project. A public deployment can be accessed at http://bestchai.bitbucket.io/shiviz/, and the source code is available at https://bitbucket.org/bestchai/shiviz/src/.

6 EVALUATION

We evaluated ShiViz in three ways, via a 39-participant controlled experiment, via a study with 70 students using ShiViz in a distributed systems class, and via two case studies, in which one developer per study used ShiViz while working on a complex system. We also evaluate XVector's performance overhead. We study four research questions, which are based around the design features of ShiViz highlighted in Section 4:

- **RQ1**: Does ShiViz help understand the relative ordering of events?
- **RQ2**: Does ShiViz help query for inter-host interaction patterns?
- **RQ3**: Does ShiViz help identify structural similarities and differences between pairs of executions?
- **RQ4**: Does ShiViz help developers to debug and verify distributed systems?

Section 6.1 describes our experiments that evaluate whether ShiViz meets its design goals (RQ1–RQ3), and Section 6.2 describes case studies of ShiViz use (RQ4). Section 6.3 evaluates XVector's overhead.

Our evaluation materials, including methodology, logs, results, and analysis, are available online [13].

6.1 Understanding Distributed Systems

To evaluate RQ1–RQ3, we designed a controlled experiment with 39 participants. We first designed a 15-question questionnaire (Figure 9) about four distributed executions of three systems: (1) an implementation of the reliable broadcast protocol (RBcast), (2) a Facebook client-server interaction (FB), (3) an enterprise distributed system called Voldemort [112] (Vold), and (4) a pair of Facebook client-server interactions (FB pair). Figure 2 shows a visualization of the Voldemort log (with some hosts hidden from view).

These questions were based on our experience teaching a fourth-year distributed systems course at the University of British Columbia. In this course, the students work in teams of 2–4 students to build complex distributed systems in Go. As they develop their systems, the students frequently run

#	Log	Question	Task	% correct (Control)	% correct (ShiViz)	p-value
1.1	RBcast	How many processes are recorded in this execution?	UI	100	92	0.51
1.2	RBcast	Are there processes in this execution that never communicate with any other process?	UI	73	92	0.18
1.3	RBcast	Which node initiates the reliable broadcast operation?	RQ1	100	96	1.0
1.4	RBcast	Which nodes execute a "Handle Tick" event?	RQ2	87	63	0.15
1.5	RBcast	How many messages does node2 send to node1?	RQ1	47	79	0.08
2.1	**FB**	**Which two hosts exhibit the request-response communication pattern in the execution?**	**RQ2**	**33**	**96**	**0.00004**
2.2	**FB**	**How many instances of the request-response communication patterns are there in this execution?**	**RQ2**	**20**	**75**	**0.001**
2.3	FB	Consider and briefly describe what you think is the role that host X plays in the Facebook system.	Semantic	60	54	0.75
2.4	**FB**	**How many threads in this execution never communicated with other threads?**	**UI**	**47**	**83**	**0.03**
3.1	Vold	Which thread generated the most number of events?	UI	73	92	0.18
3.2	Vold	Some of the events in this execution explicitly note Voldemort protocol version "vp1". How many of these events are there?	RQ2	93	92	1.0
3.3	**Vold**	**Which thread behaves most similarly to the thread named "nio-server1"?**	**Semantic**	**7**	**58**	**0.002**
3.4	**Vold**	**Which thread behaves most similarly to the thread named "vold-server2"?**	**Semantic**	**7**	**63**	**0.00007**
4.1	**FB pair**	**Consider the request-response communication pattern. Compare the number of instances of this pattern between the two executions.**	**RQ2, RQ3**	**47**	**100**	**0.0001**
4.2	FB pair	Compare the two executions graphs and notice that they look different. In your own words, explain why the shapes of the two graphs look different.	RQ3	7	13	1.0

Fig. 9. The 15 questions used in our study of how ShiViz affects understanding of distributed system executions. The ShiViz group has access to ShiViz, while the control group did not. For 6 **highlighted** questions, the ShiViz group answered correctly a statistically significantly larger fraction of understanding questions (Fischer's exact test 2-tailed, $p < .05$) than the control group. Each question relates to UI or Semantic understanding tasks, or to one of the research questions RQ1–RQ3 (Section 6). UI tasks test the effectiveness of the ShiViz UI; Semantic tasks require reasoning about the semantics of the distributed system.

	Control	Treatment
Total participants	15	24
Age (median)	22	24
Programming experience (median)	4	6
Took a networking/distributed systems course	15 (62%)	9 (60%)
Debugged a distributed system	14 (58%)	7 (47%)

Fig. 10. Demographics of the participants of the control and treatment groups in the controlled user study.

into debugging challenges and ask questions in office hours and in an online discussion forum. The questions in our study build on the student questions in the course, combined with our expertise as educators, aimed to design questions that accurately measure system comprehension.

A control group of 15 participants relied on the raw logs to answer the questions. These logs contained vector timestamps. We used raw logs as our baseline, because we know of no other log analysis tool for understanding communication patterns in distributed systems. Our baseline is what a professional developer would use today [20, 40, 85].

A treatment group of 24 participants received a 10-minute introduction to ShiViz, then used the same logs and the ShiViz tool to answer the questions. None of the participants in the treatment group had prior exposure to ShiViz. Participants in both the control and treatment groups were limited to 60 minutes. The participants were assigned to the treatment and control groups at random; this randomness led to the uneven split.

The controlled study included a mix of 6 graduate students, 8 undergraduate students, and 1 professor. The treatment study included 24 students in a combined graduate and undergraduate course: 8 graduate students and 16 undergraduate students. Figure 10 reports the demographics of the two groups.

The participants in both groups were non-experts in the field of distributed systems, though about 50% had prior experience with such systems and about 60% had taken a related course. All participants were students or employees at UMass Amherst (and none are authors of this article).

For each group and for each question, Figure 9 reports the percent of participants who answered the question correctly, and the Fisher's exact test p-value showing whether there is a statistically significant difference between the treatment and control groups. **Overall effect of using ShiViz.** First, we compared the distribution of correctly answered questions by the treatment group to that of the control group using the two-tailed unpaired t-test. The test rejected the null hypothesis that these groups came from the same distribution ($p = 0.00002$). We thus conclude that using ShiViz did significantly impact the participants' ability to answer distributed systems questions correctly. The effect size of the impact is very large (Cohen's $d = 1.56$).[5]

We then, for each question, compared the distributions of correct answers and measured if ShiViz's effect was statistically significant for that question. We used the non-parametric two-tailed Fisher's exact test that makes no assumptions about the underlying distribution and is recommended for small sample sizes. We found that for six of the fifteen questions, the test confirmed that the treatment group and control group were statistically significantly different ($p < 0.05$, **highlighted** in Figure 9). For each of these six questions, participants using ShiViz answered more questions correctly than the participants in the control group. For example, 96% of the participants using ShiViz answered question 2.1 correctly, whereas only 33% of the participants without ShiViz did so. For these six questions, on average, 52% more of the participants using ShiViz answered

[5]We use the standard effect size interpretation of Cohen's d: $d \geq .01$ is interpreted as very small, $d \geq .2$ as small, $d \geq .5$ as medium, $d \geq .8$ as large, $d \geq 1.2$ as very large, and $d \geq 2.0$ as huge [104].

questions correctly than participants in the control group. For the other questions, the difference between the groups was not statistically significant, likely an artifact of our study's size.

UI: Diagram comprehension. The treatment participants scored 83%–92% (mean 90%) on questions that tested the participants' understanding of the visual components making up the execution graph as well as the relation of these components to information captured in the input log. This was an improvement over the control group, which scored 47%–100% (mean 73%). On some questions, such as question **1.1**, the control group did better. We believe the reason for this is the lack of experience with ShiViz. To answer question **1.1** they had to simply count the number of host boxes, or timelines, in the diagram.

RQ1: Understanding event ordering. A primary purpose of the ShiViz execution graph is to help developers reason about logged events and inter-host interactions (Sections 4.1.1 and 4.1.2). Question **1.3** required identifying the host that initiated the protocol (i.e., generated an event that caused the other hosts to generate events). For this question participants had to reason about the ordering of events. Question **1.5** required a detailed inspection of the interactions between two particular hosts. Treatment participants had success rates of 96% and 79%, respectively. Control participants did well on the first question (100%), but did significantly worse on the second question (47%). Studying event order by studying the raw log is error-prone.

RQ2: Querying the execution graph. Several questions tested the participants' ability to use the ShiViz search features. Questions **1.4** and **3.2** both required performing a keyword search (Section 4.2.2) for a particular event string. However, question **1.4** also involved manual inspection of the graph for highlighted (matching) events and a correlation with the hosts that generated these events. By contrast, the control group participants easily searched the log for lines containing the keyword and the corresponding host. We think this explains why it had a lower score in the treatment group (63%) as compared to the control group (87%). For question **3.2**, however, both groups scored similarly.

Questions **2.1** and **2.2** could both be answered using the pre-defined request-response structured search (Section 4.2.1) and received respective scores of 96% and 75% in the treatment group. The control group participants scored poorly on these two questions, with an accuracy of just 33% and 20%, respectively. Even simple inter-host behavior, such as the request-response pattern used in these two questions, is difficult to identify using raw logs.

RQ3: Comparing and differencing pairs of execution graphs. At the time of this study, we had not yet implemented the feature to highlight differences between two executions, though two executions could be viewed side-by-side (Section 4.3.1). This differencing feature was partly inspired by question **4.2**, which was the most difficult question in the study, with only 12.5% and 7% of treatment and control participants, respectively, answering it correctly. This question asked the participants to reason about the divergence between two Facebook execution logs. The dissimilarity stemmed from an extra client request in one of the executions.

Most treatment participants manually compared the executions side-by-side, looking for the first point of deviation where events do not sync up between the two executions. Their answers indicated that inter-host communication in the two executions eventually occurs at different points, but nearly all participants failed to properly explain the divergence in terms of high-level system behavior. One participant presented a particularly elegant answer in the form of a user-defined structured search query (Section 4.2.1). The participant's query illustrated that one of the executions contains an additional client request.

Semantic: Conceptual understanding. Finally, we asked conceptual questions that asked participants to understand the semantics of the communication in the system. For instance, in question **2.3**, the participants were asked to describe the role of host X, which processed client requests and routed them to the appropriate data center host. Both groups did similarly on this question.

However, for questions **3.3** and **3.4**, which asked the participants to identify a host that acted similarly to another host, participants in the treatment group did significantly better (mean 61%) than those in the control group (mean 7%). The ShiViz tool helped the participants to visualize the overall host behavior, which was difficult to do using the raw, and highly detailed, logs.

Summary: Overall, we found that the participants using ShiViz had substantially higher accuracy than those participants who used the raw logs. Specifically, on 9 of the 15 questions those who used ShiViz had an accuracy score that was at least 19 absolute percentage points higher. And, Fisher's exact test shows that for 6 of the 15 questions there was statistically significant difference ($p < 0.05$) in how the two groups performed.

6.2 Developing Distributed Systems

We further conducted two studies, one with 70 students in a class (Section 6.2.1) and another with 2 developers (Section 6.2.2).

6.2.1 Distributed Systems Course Study. We conducted a study using an undergraduate distributed systems course in which 70 participants used ShiViz. None of the authors were directly involved with the instruction of this course. The course instructor presented vector clocks in a lecture and then briefly demonstrated how ShiViz can be used to visualize a log. The students were then given two homework assignments that described the file format ShiViz uses. One assignment asked the students to implement the bully leader election algorithm [16], and the other the distributed two-phase commit transaction algorithm [8]. As part of the assignments, the students were asked to write a short description of their experience using ShiViz within the assignments. Once the course ended, the instructor shared the students' assignments with us. More information about the two assignments is available online [13].

Leader election assignment. For the first assignment, the participants were required to implement a bully leader election algorithm, cause it to exhibit some interesting behavior, capture that behavior in logs, and then explain that behavior.

Many participants described how ShiViz helped them to discover, understand, and fix defects, then confirm correct behavior, in their implementation of the bully leader election algorithm. The leader election system should be robust against environmental faults such as network partitions, dropped packets, failing hosts, and slow communication links, but the initial implementations were not.

"The visualization helped us spot a bug that nodes were not sending their most current timestamps to other nodes until a COORD is sent." [Group 28] (Section 4.1.1).

"One specific instance of how ShiViz helped us debug our program was when we were struggling to understand why our timestamps seemed to get increasingly misaligned over time. It was difficult to understand the ordering of our events through our Linux console, so we used ShiViz to understand the situation." [Group 12].

"...we were noticing a problem where the second-highest node regularly set itself as the coordinator, even without any network failures. The highest node was also listing itself as a coordinator. We discovered this by examining the visualization and noticing that both the two highest nodes were often simultaneously listed as coordinators. We used this visualization to help us determine the source of the problem." [Group 19].

Almost all of the reports noted the power of visualizing the log and the usefulness of having an interactive visualization for exploring the executions.

Two-phase commit assignment. For the second assignment, the participants were required to implement a system based on the two-phase commit protocol, cause their system to complete a transaction with no errors, and then cause a transaction that commits despite the failure of a

replica during the transaction. Participants were required to explain these situations that they had caused.

In their reports, the participants frequently referred to the helpfulness of the visualization in confirming behavior: *"these ShiViz logs show a clear indication of successful functionality of the basic 2-phase commit protocols."* [Group 18]. Some groups also noted that ShiViz helped them understand concurrency in their algorithm: *"ShiViz was very useful for visualizing these parallel transactions."* [Group 15].

In both assignments, participants noted the importance of logging key events in the system, because otherwise the events are invisible to ShiViz. They also mentioned that ShiViz diagrams can be complex, particularly in cases with significant concurrency between the hosts.

6.2.2 Case Studies with Two Researchers. We performed two case studies, each one with a highly experienced systems researcher who used ShiViz to understand and debug a complex system they were working on at the time. The first researcher used ShiViz for four months while designing and debugging a distributed system for troubleshooting other systems [108]. The second researcher used ShiViz during the course of two weeks in debugging two locking implementations in the context of a commercial multi-threaded key-value store that is used as a storage engine for MongoDB [82]. In both cases, the researchers learned and used ShiViz on their own.

Case study 1: Distributed system design and debugging. The goal of the first researcher's system was to minimize input sequences to distributed systems when reproducing failures; that is, his system tries to minimize external events and internal event schedules when recreating a failed execution. In building this system, the researcher used ShiViz to debug and understand behavior across a variety of distributed systems. We performed a semi-structured interview with the researcher after the four-month period to gain insight into how he used ShiViz and what features he found most useful.

The researcher initially used ShiViz to come up with a heuristic for the minimization algorithm. Instead of enumerating the entire event schedule space to find the minimal schedule, he wanted to inspect several schedules that led to the same bug and then attempt to find unifying features among these schedules. He did this by using ShiViz to visually compare (Section 4.3.1) several event schedules from three different distributed systems: a reliable broadcast implementation, a consensus protocol called Raft [86], and a distributed hash table implementation based on Pastry [101].

The researcher also used ShiViz to debug his own system. A particular example that he described involved the debugging of a model-checking optimization called distributed partial order reduction (DPOR) [37]. He began by comparing the event schedules produced by DPOR against those generated by a simpler strategy that he knew to be correct. He found that the outputs of the two did not match, even though they should have been equivalent. To pinpoint exact areas of divergence between two such schedules, he used ShiViz to visualize outputs from both strategies and to compare the resulting execution graphs side-by-side. The generated schedules were about 600 to 700 messages long and spanned five hosts.

The side-by-side execution comparison helped the researcher efficiently locate points of deviation in the DPOR output. He noted that the visual nature of the execution graphs as well as their emphasis on the happens-before relations made it *"easy to spot where things diverge"*.[6]

Upon finding the deviation, the researcher traced the event back to the relevant statement in the original console output using the associated log lines on the left of the execution graph

[6] At the time of this study, ShiViz did not support the explicit highlighting of differences between executions (Section 4.3.1). We have added this feature in part because of this use case.

(Section 4.1.1). This allowed him to debug his implementation of DPOR without having to manually sift through the console output, a process that is both tedious and error-prone. The researcher mentioned that the console output was *"enormous and hard to understand,"* while ShiViz made the problem tractable.

As with the last use case in Section 2, the juxtaposition of two traces—one generated by the developer's implementation and one by an implementation known to be correct—enabled visual comparison, aiding in the researcher's debugging efforts. In this case, the researcher manually looked for points of divergence between the graphs, but the feature for explicitly highlighting differences facilitates this effort by immediately emphasizing any disparities. This study indicates that ShiViz is an effective tool for comparing executions and for identifying divergences in their behavior (RQ3).

Case study 2: Performance debugging of a multi-threaded system. In the second case study, a researcher, as part of her consulting work, used ShiViz for two weeks to debug performance disparities between two mutex implementations used by a multi-threaded key-value system at a small company. We performed two semi-structured interviews with the researcher to understand how she used ShiViz.

The researcher focused on improving two-lock implementation. The original version used pthread mutexes, while the new and optimized implementation used a ticket idea to provide fairness. The naive version had each thread spin in a tight loop waiting its turn. This was then optimized to have a fixed number of spinners and the remaining threads sleeping. The next (spinning) thread to grab the lock would send a signal to wake up the next sleeping thread so there was a constant number of spinners. The performance problem was that the average acquire time of a ticket-based lock was unexpectedly longer than that of a pthread mutex, even though the critical section and the time to release the lock were both shorter. This anomaly was elusive—the researcher spent a few weeks trying to understand the root cause of this problem before we approached her. She noted that existing performance debugging tools were not helpful: *"I spent several weeks trying to figure out this subtle problem using conventional performance tools that typically rely on computing averages and aggregates, and I was getting nowhere."*

To use ShiViz, the researcher first implemented an instrumentation tool to track the use of locks in each thread and to associate a fine-grained timestamp with each event. The resulting ShiViz execution graph represented the lock release/acquire events as message send/receive in the graph. To study the anomaly, the researcher generated a log with 16–30 active threads, containing about 70M events. ShiViz did not scale to visualizing the complete log, so she used ShiViz to visualize a 4K subset of events, focusing on *"trying to answer the question: what happened that made [the ticket-based lock] wait for so long? Were there unexpected delays to any events?"*

The researcher studied the timestamps associated with events in the execution graph and manually determined the time that a thread spent acquiring a lock, holding a lock, and so on. She relied heavily on ShiViz to understand the relative ordering of events between threads. She spent several hours analyzing the log with ShiViz before determining the root cause of the problem: *"With ShiViz, which enabled to see thread interactions at granularity of individual events, I figured out what was happening in a matter of hours."*

She found that for the ticket-based lock the elapsed time between one thread releasing the lock and another thread acquiring that lock was non-uniform. This was contrary to the assumption that each time the acquiring thread is spinning, as threads are sent the wake signal. The problem was that a thread may take a while to wake up: It might not wake up early enough to enter the spinning state when the lock becomes available. This explained why the lock wait times were longer than expected.

In addition to the acquisition time disparity, the researcher used ShiViz to determine that (1) one thread does different work than the others (based on the amount of time spent in this thread). By examining the code, the researcher realized that this thread is a work producer. (2) Using ShiViz, the researcher realized that the pthread mutex is unfair relative to the ticket-based lock—some threads acquire the lock more often than others when using the pthread mutex, which creates a priority inversion. And, (3) the priority inversion makes the work producer thread run more often when using the pthread mutex than when using the ticket lock. Since the work producer is the bottleneck in the studied workload, the workload ends up completing faster with the pthread mutex lock.

In summary, the researcher noted that ShiViz helped her solve the problem faster than if she used other tools: *"I seriously doubt that I would have been able to pinpoint the precise cause of the problem without ShiViz, and even if I had, it would have taken me many weeks."*

6.3 Vector Clock Instrumentation Overhead

Our XVector vector clock instrumentation libraries have (1) a network overhead due to the additional vector clock timestamps in network payloads, and (2) a vector clock logging overhead. We evaluate both of these overheads in the GoVector library, which is an XVector library for Go lang. Our experience using other XVector libraries in teaching and with open-source systems indicates that the other XVector libraries have similar overheads. We use GoVector to instrument etcd,[7] a popular distributed key value store that uses the Raft consensus protocol [87] to achieve consistency.

Experimental setup. We ran all experiments on an Intel machine with a 4-core i5 CPU and 8 GB of memory. The machine was running Ubuntu 14.04 and all applications were compiled using Go 1.6 for Linux/amd64. We exercised etcd by loading it using the Yahoo! Cloud Serving Benchmark (YCSB-B)[8] with a uniform distribution (50% gets and 50% puts on random keys). The etcd Raft cluster consisted of three replication nodes. We measured the latency and goodput[9] on both the uninstrumented and the instrumented versions of etcd. In both experiments, the maximum number of clients that could execute concurrently without reaching maximum CPU utilization was 72.

Goodput overhead. Each entry in GoVector's timestamp has two 32-bit integers: one is node identity and the other is the node's logical clock timestamp. The overhead of vector clocks is therefore a product of the number of nodes interacting during execution and the number of messages: $64\ bits \cdot nodes \cdot messages$. In practice, adding vector clocks to messages slows down each message, which can impact the behavior of the system.

Figure 11 shows the goodput for uninstrumented and instrumented Raft with each point representing an average over three executions. Adding vector clocks to Raft slowed down the broadcast of heartbeat messages and caused an *increase* in goodput with 36 clients. As the number of clients grew, the size of the vector clocks overcame this goodput saving. Across these experiments, goodput decreased by 13% on average and the maximum decrease was 23%.

Latency overhead. GoVector imposes a latency overhead in computing vector clocks and in serializing and deserializing messages with vector clocks. Figure 12 shows Raft's latency processing client requests in the uninstrumented and instrumented versions of Raft. Across these experiments, the latency increase was 7% on average with a maximum increase of 21%.

[7] https://github.com/coreos/etcd.
[8] https://github.com/brianfrankcooper/YCSB/.
[9] Goodput is application-level throughput. For etcd, it is the number of client requests that the system can process per unit time.

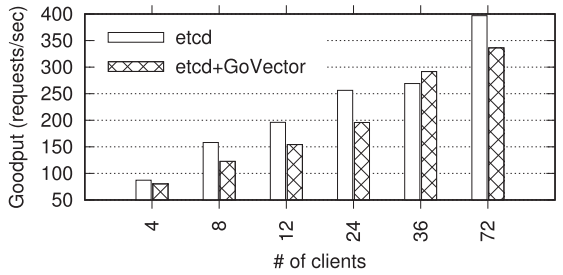

Fig. 11. Goodput of uninstrumented and instrumented etcd Raft.

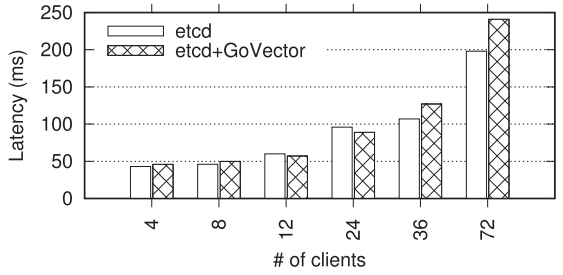

Fig. 12. Latency of uninstrumented and instrumented etcd Raft.

Logging overhead. In the above experiment, we also collected information on logging overhead. We found that the average execution time of a single logging statement is 20 microseconds. In our local area network with a round trip time of 0.05 milliseconds while running etcd with 1-second timeouts, we were able to execute approximately 50K logging statements per node before perturbing the system.

The overhead imposed by XVector makes it a viable tool during development, but not in production. We believe that existing distributed tracing systems, such as X-Trace [39] and Dapper [110], which are intended for production use, can be extended with tracking of vector-clock timestamps. We have also successfully reconstructed ShiViz-compatible logs from X-Trace traces.[10]

6.4 ShiViz Scalability on Synthetic and Benchmark Logs

ShiViz is designed to run in the browser, making it more accessible and easier to use. However, this design choice does impact its ability to scale to large and complex logs. In this section, we present scalability results from using ShiViz on a variety of logs. In each experiment, we report the time to load and visualize the log with ShiViz in a browser. In every log we experimented with, ShiViz remained real-time interactive after the visualization was rendered.

We ran all experiments on a Mac Air laptop running OS X 10.13.6 with a 2.2 GHz i7 processor and 8 GB RAM. We used a 64-bit version of the Chrome browser version 76.0.3809 with the latest version of ShiViz (commit id `4d89184c2d12`). In the first three experiments, we generated synthetic logs that had specific characteristics to allow us to control for different scalability factors. In the fourth experiment, we used logs collected from a microservices-based social network system with 36 unique microservices [42].

In the first experiment, we varied the number of events in the log. Each log in this experiment had 12 hosts; every other event in each log was a communication event (that is, 50% of the events

[10] https://github.com/MPI-SWS/xtrace-shiviz/.

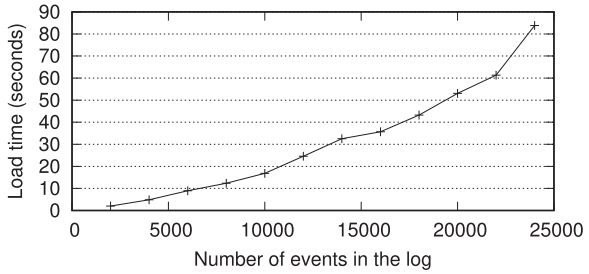

Fig. 13. ShiViz scalability for logs with varying number of total events.

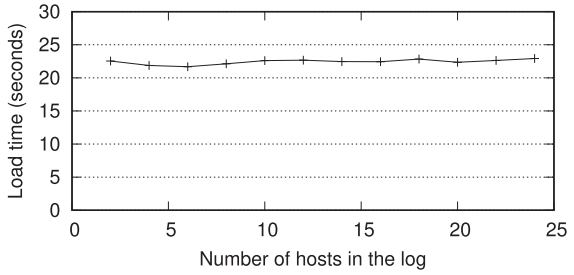

Fig. 14. ShiViz scalability for logs with varying number of hosts.

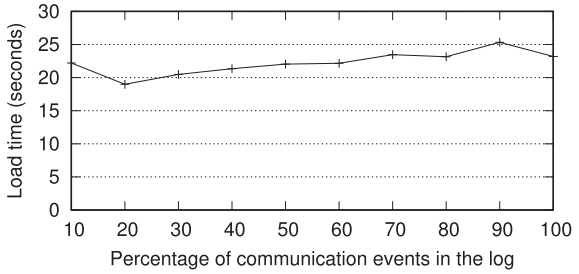

Fig. 15. ShiViz scalability for logs with varying percentage of communication events.

were communication events), and each log had between 2K and 24K total events. Figure 13 shows the time to visualize each of these logs with ShiViz. The plot illustrates that ShiViz has a nonlinear cost for processing logs in terms of the number of log events. Note, however, that ShiViz visualizes logs with fewer than 5K events in under 10 seconds. This accommodates many systems: Distributed tracing logs from the microservices-based system below contain as few as a hundred events.

In the second experiment, we varied the number of hosts in the log. Each log in this experiment had 12K events with 50% being communication events, and varied the number of hosts from 2 to 24 in steps of 2. Figure 14 shows the visualization time with ShiViz for these logs, which is around 22 seconds for all the logs. This indicates that ShiViz is not impacted by the number of hosts in the log, vector clock length (which is proportional to the number of hosts), and the number of visual host timelines that ShiViz has to generate for logs with more hosts.

In the third experiment, we varied the fraction of events in the log that were communication events. Each log in this experiment had 12K events split evenly across 12 hosts, but the fraction of communication events varied from 10% (mostly local events) to 100% (all events are communication events). Figure 15 shows the time to visualize these logs with ShiViz. The time is relatively stable,

User's interaction with system	Events in log	# Microservices	ShiViz load time (ms)	Collapsed events
Register for a new account	32	3	88	71%
Follow another user	91	7	179	62%
Unfollow another user	82	7	156	71%
Read the posts on home timeline	52	3	106	83%
Write a post on home timeline	15	2	70	87%
Read the posts on a user's timeline	63	4	123	83%
Write and upload a new post	374	22	574	77%

Fig. 16. Description of different types of interactions with the social network microservices-based application from the DeathStarBench benchmark suite [42] and the properties of the log produced by distributed tracing each of these interactions.

as the fraction of communication events varies. The cost of an extra communication edge is (1) the computation to determine which two log events to connect, and (2) an extra communication edge visual element.

The above three experiments were on synthetic logs to help us benchmark ShiViz. We also considered real logs from DeathStarBench, a recently published microservices benchmark suite by Gan et al. [42]. We used the social network system in this suite. It is composed of 36 microservices and has over 60K LOC in a variety of languages.

We used the built-in social network user interactions, such as *register for a new account*, in this application to generate the workload. We then captured the distributed trace caused by the interaction across the microservices by using an instrumented version of the benchmark with X-trace distributed tracing [39].[11] We then converted the collected distributed trace into a ShiViz-compatible log and visualized it with ShiViz.

Figure 16 describes the benchmark interactions we used to generate the logs, the size of each log, the number of microservices in each log, the time ShiViz took to visualize the log, and the fraction of events collapsed with the default collapse transformation (Figure 3(a)).

Figure 16 shows that the logs had fewer than 400 events from 2–22 microservices. ShiViz visualized each of these logs in under a second and collapsed at least 70% of the local events in these traces.[12] Based on this experience, we conclude that tracing specific parts of a complex system, such as a specific interaction in a social network system, is the most scalable way of using ShiViz.

6.5 Evaluation Summary

Overall, our evaluation suggests that ShiViz is easy to learn and to use. In particular, we found evidence to answer three of our research questions, RQ1–RQ3, in the affirmative. We also found that ShiViz is useful in a variety of contexts, including confirming system behavior, debugging, identifying anomalous behavior, and supporting system comprehension. The overhead of vector clock instrumentation depends on the system and its communication patterns; however, our experiments with etcd Raft indicate that our existing instrumentation libraries are usable and have reasonable performance in the context of complex distributed systems.

Since describing the XVector and ShiViz tools in a practitioner-oriented article [14], these tools have been used by other research groups and developers and researchers at several companies. For example, ShiViz is used in research projects focused on debugging of distributed systems that

[11] https://github.com/JonathanMace/DeathStarBench.

[12] Note that this percentage depends on the percentage of communication events in the log and is highly system-dependent.

use ShiViz to visualize the traces they produce [82, 107]. ShiViz is also used by P and P# projects within Microsoft to visualize traces of executions during debugging.[13] Akka is a popular toolkit for building concurrent and distributed systems by using actor-based programming in Java and Scala. Akka includes scripts for developers to convert their logs into ShiViz format.[14] TLC is a model checker for TLA+ and PlusCal modeling languages. TLC supports ShiViz for visualizing the counter-example trace from a model checking run of a concurrent model.[15] As a final example, the Bro network monitor includes ShiViz support to help developers visualize distributed network activity.[16]

7 THREATS TO VALIDITY

Salman et al. [102] and Pham et al. [89] have demonstrated that conclusions based on evaluations that use students can generalize to the broader developer community. We evaluated ShiViz with 70 participants in an undergraduate distributed systems course with fourth-year CS majors at the University of British Columbia, a major research university (Section 6.2.1). Most of these students became novice professional software developers only a few months later; we expect our results to generalize at least to novice engineers and perhaps beyond.

While the goal of our ShiViz evaluations was to evaluate a set of user interface features and underlying algorithms, such evaluations and interface design are inherently iterative. The developers' interactions with the tool inform the tool makers and suggest new interface features. For example, our studies revealed that developers might benefit from highlighting differences between two executions, as described in Section 4.3.1. We elected to implement this feature as part of ShiViz and include the description of this finding here, even though the ShiViz version used for evaluation did not yet include this feature: ShiViz displayed executions side-by-side but did not highlight differences. Adding such features could, in theory, alter our results, although the effect is unlikely to be significant.

Our study asked participants to answer questions (Figure 9) about system behavior. The goal of the questions is to gauge the correctness of the participants' understanding of the system behavior. We used our expertise as educators to design questions that accurately measure system comprehension. More questions would likely have resulted in more accurate results, but the study had to balance this accuracy against the length of the study.

8 DISCUSSION

ShiViz surfaces low-level ordering information, which makes it a poor choice for understanding aggregate system behavior. The ShiViz visualization is based on logical and not real-time ordering and cannot be used to study certain performance characteristics. The ShiViz tool is implemented as a client-side-only browser application, making it portable and appropriate for analyzing sensitive log data. However, this design choice also limits its scalability.

ShiViz reconstructs the happens-before relation using vector timestamps associated with logged events. The protocol for maintaining these timestamps (detailed in Section 3.2) does not provide ordering information when messages are lost. In particular, a send event without a corresponding receive event will not be identified as a send event. This is because the event that immediately happens-after the send event cannot be associated with a remote host, since the sent message was lost.

[13] https://github.com/p-org/TraceVisualizer.
[14] https://github.com/akka/akka-logging-convertor.
[15] https://github.com/tlaplus/tlaplus/issues/267.
[16] https://www.bro.org/bro4pros2017/Charousset_CAF_Bro4Pros2017.pdf.

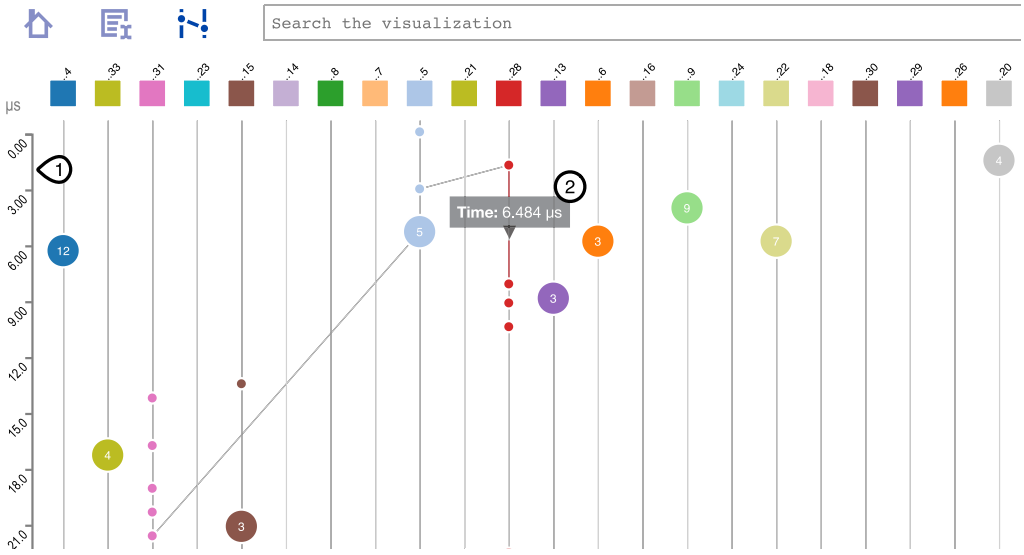

Fig. 17. A screenshot of TSViz, a tool that extends ShiViz to support real-time timestamps. ① A global clock time scale totally orders all events in the log. ② The user can interact with not only the visualized events, but also with the intervals between events.

Optimizing visualization layout. At the moment, ShiViz presents hosts in a set order: either ordering them from left to right by the number of events each host generated in total, or by their appearance in the log. We are working to formulate an algorithm that can produce a host ordering that minimizes the total number of line crossings (this problem is NP-hard).

Support for real-time timestamps with TSViz. We have previously described TSViz, a tool based on ShiViz that focuses on logs with real-time timestamps in a short tool-demonstration paper [83]. TSViz uses logs that are composed of both logical and real-time timestamps to give developers a better sense of when events occur relative to each other. This is in contrast to ShiViz, which places each event host on a process timeline as soon as possible without violating the happens-before requirements and using a default minimum spacing between events. Figure 17 shows a screenshot of TSviz.

We envision a number of additional features to make ShiViz even more useful to developers.

Real-time online visualizations. Rather than wait for an execution to complete and then collect logs from each process for visualization with ShiViz, we could augment XVector to automatically upload logged events to the ShiViz website as they are generated. ShiViz would then generate a scrolling visualization of the system in real-time.

Linking the visualization with code. An event in a ShiViz visualization corresponds to some program point that generated the event. A developer interested in jumping to this program point from the visualization must manually find and jump to the program point. This process could be automated and the visualization could even be integrated into IDEs.

9 RELATED WORK

We have briefly described ShiViz in two practitioner-oriented articles that overview the space of approaches to debugging distributed systems [14] and as a poster [1]. None of these prior publications present ShiViz's technical contributions and none evaluate ShiViz.

The rest of this section reviews prior work in the space of log analysis, visualization, and tracing, with a special focus on distributed systems.

9.1 Analysis of Totally Ordered Logs

There are many log analysis tools for processing totally ordered logs, including fluentd [38], logstash [76], graylog [47], splunk [111], papertrail [87], loggly [75], sumologic [113], and Zinsight [26, 27]. A common enterprise solution for end-to-end log analysis is the ELK stack for indexing and storing traces (elasticsearch [34]), filtering and extracting data from traces (logstash [76]), and visualization of the results (kibana [60]).

These tools can aid manual performance analysis and debugging [26], such as the detection of memory leaks in Java [30], and can be applied to the setting of streaming logs [25, 29], even at scale [24].

Unlike ShiViz, the goal of these tools, including the ELK stack, excludes capturing and analyzing partial ordering information. As a result, unlike ShiViz, these tools do not help developers understand the exact ordering of events in a system with concurrent processes.

9.2 Visualizing Distributed Systems

The most closely related systems to ShiViz are Poet [65] and DTVS [33]. Poet is a toolchain for instrumenting and then visualizing distributed systems as time-space diagrams. DTVS visualizes concurrency regions in a trace of synchronous distributed system executions. ShiViz makes several technical advances over both Poet and DTVS, including graph transformations (e.g., filter by host), keyword and structured search, multi-execution comparison, and clustering. ShiViz is also simpler to use and deploy: ShiViz does not require a server, runs in a browser, and requires no installation by the developer. The Poet and DTVS tools have **not been evaluated with developers**. By contrast, we extensively evaluated ShiViz in several studies with over 100 developers.

Hy+ [21] is a system for parallel and distributed systems debugging. Hy+ uses a visualization technique that is a cross between Harel's statecharts [52] and directed hypergraphs. ShiViz's structured search feature comes from Hy+, which allows developers to use the GraphLog language to specify communication patterns of interest. However, unlike ShiViz, Hy+ does not allow developers to compare and cluster multiple executions.

Pajé [18, 19] is a tool to instrument a multi-threaded system to produce partially ordered traces. The tool provides features such as the ability to rewind execution or pinpoint the source code line that generated a specific event. Pajé does not support ShiViz's features such as keyword search, structured search, and comparison and differencing of multiple executions.

De Pauw et al. mine communication flow patterns between web services at transaction granularity and have developed a visualization tool to cluster and present the pattern clusters to developers [28]. The tool visualizes web-server interaction patterns as a partial order. It does not, however, support structured search, execution comparison, or associating the diagram with logged information.

Visualizing communication patterns in publish/subscribe systems can help developers and administrators understand and manage topic-specific message flows and their performance. The graph-like visualizations can be overlaid on maps to help understand spatial and geographical relationships in message flows [59].

For Charm++ programs, it is possible to use heuristics to partially recover partial order, compensating for missing dependencies [54]. ShiViz and XVector are more general and apply to programs other than Charm++ programs, but ShiViz requires logged vector clock timestamps for precisely reconstructing partial order, while XVector may require developers to modify their code.

Ravel [55, 56] is a tool to scalably visualize the communication topologies of MPI applications with thousands of processes. Ravel visualizes both the partial order and the physical clock order of MPI send and receive events. Ravel has multiple features to scale the visualization to very large traces, such as strategies to cluster processes and their communication patterns. By contrast, ShiViz is a *log* analysis tool whose visualization is tightly linked to the input text log. The two tools are complementary: ShiViz has features that Ravel does not, including keyword search, structured search, and comparison and differencing of multiple executions. And, Ravel includes features for understanding performance that are not present in ShiViz. Our ongoing work on TSViz incorporates some features to visualize performance into a ShiViz-like interface [83] (as discussed in Section 8).

DAGViz [53] can visualize execution traces of programs written in a task-based parallel programming paradigm for understanding the performance of such problems. Task-based parallel programming, by design, hides the runtime scheduling mechanisms from the developer, taking performance-affecting decisions out of the developers' hands. DAGViz aids understanding of performance bottlenecks. ShiViz could also be used to visualize task-based parallel programs, which can similarly help understand bottlenecks, although ShiViz's focus is not performance. Unlike DAGViz, ShiViz is general and can be used to visualize other distributed programs than task-based parallel ones.

Two-dimensional and three-dimensional visualizations can help understand the network traffic patterns of massively parallel programs, e.g., those that execute on supercomputers. Such visualizations embed the physical network topology and can help understand and improve system performance [69]. Logical time can help create these large-scale visualizations [55]. By contrast, ShiViz does not focus on the physical topology, and hosts in the ShiViz visualizations can be co-located on the same physical machine or be distributed on different machines. ShiViz also focuses on relatively smaller systems than the massively parallel ones capable of executing on supercomputers.

Comparing executions can be difficult if the executions are long. ShiViz allows users to filter events and executions to reduce the visual load. Other techniques, such as aggregating the events or data [105], visualizing heatmaps [7, 67], bundling edges [118], or animating the visualization [70] can also manage long executions. Inherently, these techniques aim to abstract away some details of the execution to make it possible to visualize a larger execution, while ShiViz aims to show as many details as is reasonable to help developers understand the relevant behavior. Such work is complementary to ours and ShiViz could be extended with such approaches for long executions; however, such extensions are beyond the scope of this article.

9.3 Visualizing Distributed Systems without a Partial Order

Distributed system behavior can be visualized without representing the partial ordering of events. Theia [43] displays a visual signature that summarizes various aspects of a Hadoop execution, such as the execution's resource utilization. Similarly, shared resource use can be modeled and visualized to represent complex system behavior, such as MapReduce and WatsonDeepQA [31]. SALSA [116] and Mochi [117] are log analysis tools that extract and visualize the control- and data-flow of Hadoop applications' logic in a variety of forms, including finite state machines. These tools can be used for failure diagnosis and performance debugging. Overall, tools in this category necessarily provide high-level summaries of a system's behavior. ViVA [73] has a similar goal as ShiViz, but its focus is on visualizing component interactions in a distributed setting; ViVA neither tracks nor shows the partial ordering of events.

A promising alternative approach is to use NLP techniques to reconstruct distributed workflow information from existing logs, e.g., as in recent work by Pi et al. [90]. This approach does not

require logical clock instrumentation, but the accuracy of the final result is more appropriate for coarse-grained analyses.

9.4 Tracing and Debugging Distributed Systems

A variety of tracing systems have been proposed to track requests in distributed systems, such as Dapper [110], Zipkin [125], X-Trace [39], and others [3, 5, 100, 115]. For example, vPath [115] is a VM monitor for understanding causality by tracking thread and network activity, and WAP5 [103] relinks a target application to custom system libraries to monitor network communication and reconstruct request paths using statistical inference. By contrast, XVector uses vector clocks to reconstruct the exact partial ordering of events in the system without singling out requests.

ShiViz's side-by-side view with differencing is similar to the side-by-side view in work to evaluate the visualization strategies for distributed request-flow comparison by Sambasivan et al. [103]. However, request flow visualization is fundamentally different from observing the partial ordering of *all* events in the system. In particular, a single request flow rarely has the same amount of concurrency. More recent work on differencing execution traces [114] collapses loops and clusters traces; adding these features to ShiViz is likely useful.

XVector adds useful partial ordering information to execution logs. We have previously used this information to mine temporal properties to relate events at different hosts and to infer a model of the distributed system from multiple executions. These models, too, were useful for understanding system behavior [10, 11].

Reproducing a bug is a key aspect of debugging. Unfortunately, reproducing bugs in distributed systems can be challenging due to sources of nondeterminism, including scheduling and parallel execution. Friday [44] helps replay distributed system executions, eliminating some sources of nondeterminism. Recon [72] can further aid debugging by collecting relations between distributed system artifacts, such as hosts, communication channels, events, and so on, and enabling querying these relationships after the execution. The goal of ShiViz is to improve understanding of a distributed system execution, which is often a necessary step in debugging distributed systems. ShiViz does not directly facilitate execution replay, although the happens-before relationship XVector logs can help reconstruct what happened in a non-deterministic execution. The work on debugging distributed systems is complementary to ShiViz, and these tools may benefit from ShiViz visualizations in conveying system behavior to the developer.

DistIA [17] is a dynamic impact analysis approach for distributed systems. This tool does not visualize/explain executions as ShiViz does; instead, DistIA helps developers make sense of how changes to their code would influence the execution of the processes in the distributed system.

9.5 Summarizing Logs and System Behavior

Numerous techniques attempt to summarize multiple system executions as a behavioral model. These approaches target both sequential executions [9, 12, 15, 22, 35, 41, 45, 62, 63, 71, 74, 77, 84, 98, 106, 119] as well as concurrent executions [2, 10, 64]. Such summaries can be created remotely and used to facilitate code reuse [78, 79]. The goal of summarization is to abstract the observed behavior, which is necessarily partial, into a compact representation that often includes other, unobserved, but likely behavior. As a result, while these techniques strive to abstract the behavior precisely, they often make overzealous abstraction decisions that result in loss of precision [62, 71]. Unlike these summarization techniques, ShiViz focuses on visualizing executions exactly as they occurred. ShiViz does not attempt to predict unobserved behavior.

9.6 Performance-based Analysis

Distributed system visualization can help understand system performance, and many existing tools aim to analyze or visualize performance-related aspects of distributed system behavior [58, 59]. Coz [23] finds causal performance relationships between execution blocks in concurrent systems. Coz can be used to identify how speeding up a particular block or set of blocks will affect system running time. Unlike ShiViz, Coz does not visualize system executions but does help developers understand relationships between particular kinds of system events.

Flame graphs [48, 49] visualize profiler output to help developers reason about how software is consuming resources, which can speed up root cause analysis. Flame graphs combine common elements using a resource, using element hierarchy to visualize those elements that use the most resources. These graphs can be explored interactively to understand resource use.

9.7 Integrated Development Environments

Visualizations of system implementation source code, control flow, data flow, and so on, have long been used to help developers understand and debug their systems. One such common use of visualizations is by the integrated development environment (IDE). PECAN [91] supports semantic views of a program expression trees, data type diagrams, flow graphs, and the symbol table. The Field environment [93] combines multiple views of the code itself. The Garden environment [92] allows combining graphical and code-based visualizations and supports customizations to the code visualizations to better align with the developer's workflow. The Desert environment [95] allows integrating multiple tools into one environment. Cacti [94] allows combining multiple data sources, such as static and dynamic analyses, into visualizations in an IDE. JIVE [97, 99] and JOVE [99] allow the developer to visualize dynamic analysis traces to quickly understand the implications of changes to code. BLOOM [96] provides a toolkit for data collection, data analysis, and data visualization, again, to help developers understand particular aspects of their systems' source code through static and dynamic analyses, aiding development and debugging. OverView [32] focuses specifically on visualizing distributed systems in the IDE, helping developers design distributed protocols by creating appropriate abstractions. While these IDEs and IDE components all help visualize various aspects of software systems, they are each quite distinct from ShiViz and its goal of visualizing concurrency of actual executions via the partial ordering of events in a distributed execution.

10 CONTRIBUTIONS

Logging is a common debugging technique, though the logs generated by distributed systems are often difficult to reason about manually. This article introduced a new method to help developers reason about distributed system executions. This method consists of XVector, to automate logging of concurrent behavior in distributed systems, and ShiViz, to visualize the logged executions.

Our method helps both novice and advanced developers to gain insight into the operation and design of their distributed systems. Our evaluation case studies suggest that ShiViz is effective at communicating execution information to the developers and that its visualization transformations and querying mechanisms are helpful. ShiViz helps developers detect issues with their systems and increases the developers' confidence in the correctness of their systems.

In addition to developing this method for reasoning about distributed system executions, we implemented XVector as a suite of libraries to augment distributed system execution logs with partial ordering information and ShiViz as a browser-based tool. All our tools, experimental design, and data are available publicly. The XVector libraries source code is available at

https://github.com/DistributedClocks. The ShiViz source code and a public ShiViz deployment are available at http://bestchai.bitbucket.io/shiviz/.

ACKNOWLEDGMENTS

We thank Jenny Abrahamson for developing the initial XVector and ShiViz prototypes and Vaastav Anand, who helped us evaluate ShiViz' scalability and also helped with development of several XVector variants. We also thank Graham St-Laurent and Matheus Nunes for making numerous improvements to the ShiViz implementation. Further, we thank to Donald Acton and Colin Scott, who helped us evaluate ShiViz, as well as all the participants in our user studies.

REFERENCES

[1] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D. Ernst. 2014. Shedding light on distributed system executions. In *Proceedings of the International Conference on Software Engineering (ICSE Poster track)* (4–6). 598–599. DOI: https://doi.org/10.1145/2591062.2591134

[2] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'07)*.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. *SIGOPS OSR* 37, 5 (2003), 74–89.

[4] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. 2008. Interval tree clocks. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'08)*. Springer-Verlag, 259–274. DOI: https://doi.org/10.1007/978-3-540-92221-6_18

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling. In *Proceedings of the Symposium on Operating Systems Design & Implementation (OSDI'04)*. 259–272.

[6] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. 2009. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Comput.* 35, 12 (2009), 595–607.

[7] Omar Benomar, Houari Sahraoui, and Pierre Poulin. 2013. Visualizing software dynamicities with heat maps. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT'13)*. DOI: https://doi.org/10.1109/VISSOFT.2013.6650524

[8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems (Chapter 7)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[9] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2015. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. Softw. Eng.* 41, 4 (Apr. 2015), 408–428. DOI: https://doi.org/10.1109/TSE.2014.2369047

[10] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. 468–479.

[11] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. 2012. Mining temporal invariants from partially ordered logs. *SIGOPS Oper. Syst. Rev.* 45, 3 (Jan. 2012), 39–46. DOI: https://doi.org/10.1145/2094091.2094101

[12] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE'11)*. 267–277.

[13] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2017. ShiViz evaluation details. Retrieved from http://bestchai.bitbucket.io/shiviz-evaluation/.

[14] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2016. Debugging distributed systems. *Commun. ACM* 59, 8 (Aug. 2016), 32–37. DOI: https://doi.org/10.1145/2909480

[15] Alan W. Biermann and Jerome A. Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.* 21, 6 (1972), 592–597.

[16] Bully algorithm 2015. Retrieved from http://en.wikipedia.org/wiki/Bully_algorithm.

[17] Haipeng Cai and Douglas Thain. 2016. DistIA: A cost-effective dynamic impact analysis for distributed programs. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*.

[18] Jacques Chassin de Kergommeaux and Benhur de Oliveira Stein. 2003. Flexible performance visualization of parallel and distributed applications. *Fut. Gen. Comput. Syst.* 19, 5 (2003), 735–747.

[19] Jacques Chassin de Kergommeaux, Benhur de Oliveira Stein, and Pierre-Eric Bernard. 2000. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Comput.* 26, 10 (2000), 1253–1274.

[20] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing logging practices in Java-based open source software projects—A replication study in Apache Software Foundation. *Empir. Softw. Eng.* 22, 1 (Feb. 2017), 330–374. DOI: https://doi.org/10.1007/s10664-016-9429-5

[21] Mariano C. Consens, Masum Z. Hasan, and Alberto O. Mendelzon. 1993. Debugging distributed programs by visualizing and querying event traces. In *Proceedings of the Conference on Applications of Databases*, Vol. 819. 181–183.

[22] Jonathan E. Cook and Alexander L. Wolf. 1998. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.* 7, 3 (1998).

[23] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'15)*. 184–197. DOI: https://doi.org/10.1145/2815400.2815409

[24] Wim De Pauw and Henrique Andrade. 2009. Visualizing large-scale streaming applications. *Inf. Vis.* 8, 2 (Apr. 2009), 87–106. DOI: https://doi.org/10.1057/ivs.2009.5

[25] Wim De Pauw, Henrique Andrade, and Lisa Amini. 2008. StreamSight: A visualization tool for large-scale streaming applications. In *Proceedings of the International Symposium on Software Visualization (SoftVis'08)*. 125–134. DOI: https://doi.org/10.1145/1409720.1409741

[26] Wim De Pauw and Steve Heisig. 2010. Visual and algorithmic tooling for system trace analysis: A case study. *Oper. Syst. Rev.* 44, 1 (2010), 97–102. DOI: https://doi.org/10.1145/1740390.1740412

[27] Wim De Pauw and Steve Heisig. 2010. Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the International Symposium on Software Visualization (SoftVis'10)*. 143–152.

[28] Wim De Pauw, Sophia Krasikov, and John F. Morar. 2006. Execution patterns for visualizing web services. In *Proceedings of the International Symposium on Software Visualization (SoftVis'06)*. 37–45.

[29] Wim De Pauw, Mihai Letia, Bugra Gedik, Henrique Andrade, Andy Frenkiel, Michael Pfeifer, and Daby M. Sow. 2010. Visual debugging for stream processing applications. In *Proceedings of the International Conference on Runtime Verification (RV'10)*. 18–35.

[30] Wim De Pauw and John M. Vlissides. 1998. Visualizing object-oriented programs with Jinsight. In *Proceedings of the Workshop Ion on Object-Oriented Technology*. 541–542.

[31] Wim De Pauw, Joel L. Wolf, and Andrey Balmin. 2013. Visualizing jobs with shared resources in distributed environments. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT'13)*. DOI: https://doi.org/10.1109/VISSOFT.2013.6650535

[32] Travis Desell, Harihar Narasimha Iyer, Carlos Varela, and Abe Stephens. 2004. OverView: A framework for generic online visualization of distributed systems. *Electron. Notes Theor. Comput. Sci. (Eclipse Technol. Exch.: eTX Eclipse Phenom.)* 107 (2004), 87–101. DOI: https://doi.org/10.1016/j.entcs.2004.02.050 2004.

[33] Dennis Edwards and Phil Kearns. 1994. DTVS: A distributed trace visualization system. In *Proceedings of the Symposium on Parallel and Distributed Processing (IPDPS'94)*. 281–288.

[34] elasticsearch 2016. Retrieved from https://www.elastic.co/products/elasticsearch.

[35] Dirk Fahland, David Lo, and Shahar Maoz. 2013. Mining branching-time scenarios. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*.

[36] Colin J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Australasian Computer Science Conference (ACSC'88)*. 55–66.

[37] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. 110–121.

[38] fluentd 2016. Retrieved from http://www.fluentd.org/.

[39] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A pervasive network tracing framework. In *Proceedings of the USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. 271–284.

[40] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? An empirical study on logging practices in industry. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*.

[41] Mark Gabel and Zhendong Su. 2008. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE'08)*. DOI: https://doi.org/10.1145/1453101.1453150

[42] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[43] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2012. Theia: Visual signatures for problem diagnosis in large hadoop clusters. In *Proceedings of the International Conference on Large Installation System Administration: Strategies, Tools, and Techniques (LISA'12)*. 33–42.

[44] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2007. Friday: Global comprehension for distributed replay. In *Proceedings of the USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. 21–21.

[45] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. 2014. Mining behavior models from user-intensive web applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'14)*.

[46] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and asserting distributed system invariants. In *Proceedings of the International Conference on Software Engineering (ICSE'18)*. 1149–1159. DOI: https://doi.org/10.1145/3180155.3180199

[47] graylog. 2016. Retrieved from https://www.graylog.org/.

[48] Brendan Gregg. 2016. The flame graph. *Commun. ACM* 59, 6 (June 2016), 48–57. DOI: https://doi.org/10.1145/2909476

[49] Brendan Gregg. 2017. Visualizing performance with flame graphs. In *Proceedings of the USENIX Annual Technical Conference (ATC'17)*. 81–88.

[50] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-Anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*.

[51] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'16)*.

[52] David Harel. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.* 8, 3 (1987), 231–274.

[53] An Huynh, Douglas Thain, Miquel Pericàs, and Kenjiro Taura. 2015. DAGViz: A DAG visualization tool for analyzing task-parallel program traces. In *Proceedings of the 2nd Workshop on Visual Performance Analysis (VPA'15)*. 3:1–3:8. DOI: https://doi.org/10.1145/2835238.2835241

[54] Katherine E. Isaacs, Abhinav Bhatele, Jonathan Lifflander, David Böhme, Todd Gamblin, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. 2015. Recovering logical structure from Charm++ event traces. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. 49:1–49:12. DOI: https://doi.org/10.1145/2807591.2807634

[55] Katherine E. Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. 2014. Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE Trans. Vis. Comput. Graph.* 20, 12 (Dec. 2014), 2349–2358. DOI: https://doi.org/10.1109/TVCG.2014.2346456

[56] Katherine E. Isaacs, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. 2016. Ordering traces logically to identify lateness in message passing programs. *IEEE Trans. Parallel Distrib. Syst.* 27, 3 (Mar. 2016), 829–840. DOI: https://doi.org/10.1109/TPDS.2015.2417531

[57] Katherine E. Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. 2014. State-of-the-art of performance visualization. In *Proceedings of the Eurographics Conference on Visualization (EuroVis'14)*.

[58] Hank Jakiela. 1995. Performance visualization of a distributed system: A case study. *Computer* 28, 11 (Nov. 1995), 30–36. DOI: https://doi.org/10.1109/2.471177

[59] Kyriakos Karenos, Wim De Pauw, and Hui Lei. 2011. A topic-based visualization tool for distributed publish/subscribe messaging. In *Proceedings of the International Symposium on Applications and the Internet (SAINT'11)*. 65–74. DOI: https://doi.org/10.1109/SAINT.2011.19.

[60] kibana 2016. Retrieved from https://www.elastic.co/products/kibana.

[61] J. Klensin. 2008. Simple Mail Transfer Protocol. *RFC 5321 (Draft Standard)*. Retrieved from http://www.ietf.org/rfc/rfc5321.txt.

[62] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)* (16–22). 178–189. DOI: https://doi.org/10.1145/2635868.2635890

[63] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. 2010. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the International Conference on Software Engineering (ICSE NIER track)* (2–8). 179–182. DOI: https://doi.org/10.1145/1810295.1810324

[64] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. 2012. Inferring class level specifications for distributed systems. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'12)*.

[65] Thomas Kunz, David J. Taylor, and James P. Black. 1997. Poet: Target-system independent visualizations of complex distributed-application executions. *Comput. J.* 1 (1997), 452–461.

[66] James F. Kurose and Keith W. Ross. 2012. *Computer Networking: A Top-down Approach* (6th ed.). Pearson.
[67] Fabrizio Lamberti and Gianluca Paravati. 2015. VDHM: Viewport-DOM based heat maps as a tool for visually aggregating web users' interaction data from mobile and heterogeneous devices. In *Proceedings of the IEEE International Conference on Mobile Services (MS'15)*. 33–40. DOI: https://doi.org/10.1109/MobServ.2015.15
[68] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
[69] Aaditya G. Landge, Joshua A. Levine, Katherine E. Isaacs, Abhinav Bhatele, Todd Gamblin, Martin Schulz, Steve H. Langer, Peer-Timo Bremer,andValerio Pascucci. 2012. Visualizing network traffic to understand the performance of massively parallel simulations. *IEEE Trans. Vis. Comput. Graph.* 18, 12 (Dec. 2012), 2467–2476. DOI: https://doi.org/10.1109/TVCG.2012.286
[70] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. 2008. Exploring the evolution of software quality with animated visualization. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'08)*. 13–20. DOI: https://doi.org/10.1109/VLHCC.2008.4639052
[71] Tien-Duy B. Le, Xuan-Bach D. Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing specification miners through model fissions and fusions. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. 115–125. DOI: https://doi.org/10.1109/ASE.2015.83
[72] Kyu Hyung Lee, Nick Sumner, Xiangyu Zhang, and Patrick Eugster. 2011. Unified debugging of distributed systems with Recon. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN'11)*. 85–96. DOI: https://doi.org/10.1109/DSN.2011.5958209
[73] Youn Kyu Lee, Jae Young Bang, Joshua Garcia, and Nenad Medvidovic. 2014. ViVA: A visualization and analysis tool for distributed event-based systems. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE Demo track)*. 580–583.
[74] David Lo and Shahar Maoz. 2010. Scenario-based and value-based specification mining: Better together. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. 387–396.
[75] loggly 2016. Retrieved from https://www.loggly.com/.
[76] logstash 2016. Retrieved from https://www.elastic.co/products/logstash.
[77] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*. 501–510. DOI: https://doi.org/10.1145/1368088.1368157
[78] Stuart Marshall, Kirk Jackson, Craig Anslow, and Robert Biddle. 2003. Aspects to visualising reusable components. In *Proceedings of the Asia-Pacific Symposium on Information Visualisation (APVis'03)*, Vol. 24. 81–88.
[79] Stuart Marshall, Kirk Jackson, Robert Biddle, Michael McGavin, Ewan Tempero, and Matthew Duignan. 2001. Visualising reusable software over the web. In *Proceedings of the Asia-Pacific Symposium on Information Visualisation (APVis'01)*, Vol. 9. 103–111.
[80] Friedemann Mattern. 1989. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.
[81] mongodb 2016. Retrieved from https://www.mongodb.org/.
[82] F. Neves, N. Machado, and J. Pereira. 2018. Falcon: A practical log-based analysis tool for distributed systems. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*. 534–541. DOI: https://doi.org/10.1109/DSN.2018.00061
[83] Matheus Nunes, Ashaya Sharma Harjeet Lalh, Augustine Wong, Svetozar Miucin, Alexandra Fedorova, and Ivan Beschastnikh. 2017. Studying multi-threaded behavior with TSViz. In *Proceedings of the International Conference on Software Engineering (ICSE Demo track)*.
[84] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. 2014. Behavioral resource-aware model inference. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'14)* (15–19). 19–30. DOI: https://doi.org/10.1145/2642937.2642988
[85] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *Commun. ACM* 55, 2 (Feb. 2012), 55–61. DOI: https://doi.org/10.1145/2076450.2076466
[86] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*. 305–320.
[87] papertrailapp 2016. Retrieved from https://papertrailapp.com/.
[88] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *Proceedings of the International Conference on Software Engineering (ICSE'15)*.
[89] Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. 2014. Enablers, inhibitors, and perceptions of testing in novice software teams. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*.

[90] Aidi Pi, Wei Chen, Shaoqi Wang, and Xiaobo Zhou. 2019. Semantic-aware workflow construction and analysis for distributed data analytics systems. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC'19)*. ACM, New York, NY. DOI: https://doi.org/10.1145/3307681.3325404

[91] Steven P. Reiss. 1985. PECAN: Program development systems that support multiple views. *IEEE Trans. Softw. Eng.* 11, 3 (1985), 276–285. DOI: https://doi.org/10.1109/TSE.1985.232211

[92] Steven P. Reiss. 1987. Working in the garden environment for conceptual programming. *IEEE Softw.* 4, 6 (1987), 16–27. DOI: https://doi.org/10.1109/MS.1987.231801

[93] Steven P. Reiss. 1990. Connecting tools using message passing in the field environment. *IEEE Softw.* 7, 4 (1990), 57–66. DOI: https://doi.org/10.1109/52.56450

[94] Steven P. Reiss. 1997. Cacti: A front end for program visualization. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis'97)*. 46–49. DOI: https://doi.org/10.1109/INFVIS.1997.636785

[95] Steven P. Reiss. 1999. The desert environment. *ACM Trans. Softw. Eng. Methodol.* 8, 4 (1999), 297–342. DOI: https://doi.org/Reiss97

[96] Steven P. Reiss. 2001. An overview of BLOOM. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. 2–5. DOI: https://doi.org/10.1145/379605.379629

[97] Steven P. Reiss. 2003. JIVE: Visualizing Java in action demonstration description. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'03)*. 820–821. DOI: https://doi.org/10.1109/ICSE.2003.1201303

[98] Steven P. Reiss and Manos Renieris. 2001. Encoding program executions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'01)*. 221–230.

[99] Steven P. Reiss and Manos Renieris. 2005. Demonstration of JIVE and JOVE: Java as it happens. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE Demo Track)*. 662–663. DOI: https://doi.org/10.1145/1062455.1062597

[100] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. 2006. WAP5: Black-box performance debugging for wide-area systems. In *Proceedings of the International Conference on World Wide Web (WWW'06)*. DOI: https://doi.org/10.1145/1135777.1135830

[101] Antony I. T. Rowstron and Peter Druschel. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'01)*. 329–350.

[102] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments? In *Proceedings of the International Conference on Software Engineering (ICSE'15)*.

[103] R. R. Sambasivan, I. Shafer, M. L. Mazurek, and G. R. Ganger. 2013. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE Trans. Vis. Comput. Graph.* 19, 12 (Dec. 2013), 2466–2475. DOI: https://doi.org/10.1109/TVCG.2013.233

[104] Shlomo S. Sawilowsky. 2009. New effect size rules of thumb. *J. Mod. Appl. Stat. Meth.* 8, 2 (2009), 467-474.

[105] Teseo Schneider, Yuriy Tymchuk, Ronie Salgado, and Alexandre Bergel. 2016. CuboidMatrix: Exploring dynamic structural connections in software components using space-time cube. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT'16)*. 116–125. DOI: https://doi.org/10.1109/VISSOFT.2016.17

[106] Matthias Schur, Andreas Roth, and Andreas Zeller. 2013. Mining behavior models from enterprise web applications. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'13)*. 422–432.

[107] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. 2016. Minimizing faulty executions of distributed systems. In *Proceedings of the USENIX Conference on Networked Systems Design & Implementation (NSDI'16)*. 291–309.

[108] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H. B. Acharya, Kyriakos Zarifis, and Scott Shenker. 2014. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'14)*. 395–406.

[109] Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan, and Zhen Ming Jiang. 2014. Understanding log lines using development knowledge. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14)*. 21–30. DOI: https://doi.org/10.1109/ICSME.2014.24

[110] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. Retrieved from http://research.google.com/archive/papers/dapper-2010-1.pdf.

[111] splunk 2016. Retrieved from http://www.splunk.com/.

[112] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. 2012. Serving large-scale batch computed data with Project Voldemort. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'12)*. 18–18.

[113] sumologic 2016. Retrieved from http://www.sumologic.com/.
[114] Saeed Taheri, Ian Briggs, Martin Burtscher, and Ganesh Gopalakrishnan. 2019. DiffTrace: Efficient whole-program trace analysis and diffing for debugging. In *Proceedings of the International Conference on Cluster Computing*. IEEE.
[115] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. 2009. vPath: Precise discovery of request processing paths from black-box observations of thread and network activities. In *Proceedings of the USENIX Annual Technical Conference (USENIX'09)*. 19:1–19:14.
[116] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. 2008. SALSA: analyzing logs as state machines. In *Proceedings of the 1st USENIX Conference on Analysis of System Logs (WASL'08)*.
[117] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. 2009. Mochi: Visual log-analysis based tools for debugging hadoop. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'09)*.
[118] Jonas Trümper, Jürgen Döllner, and Alexandru Telea. 2013. Multiscale visual comparison of execution traces. In *Proceedings of the International Conference on Program Comprehension (ICPC'13)*. 53–62. DOI: https://doi.org/10.1109/ICPC.2013.6613833
[119] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring finite-state models with temporal constraints. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. 248–257.
[120] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. 2017. How do system administrators resolve access-denied issues in the real world? In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'17)*.
[121] Tianyin Xu and Yuanyuan Zhou. 2015. Systems approaches to tackling configuration errors: A survey. *Comput. Surv.* 47, 4 (July 2015), 70:1–70:41. DOI: https://doi.org/10.1145/2791577
[122] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2010. Experience mining Google's production console logs. In *Proceedings of the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML'10)*.
[123] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*.
[124] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'17)*.
[125] Zipkin 2016. Retrieved from http://zipkin.io/.

Received October 2018; revised September 2019; accepted November 2019