TEXT ONLY

Debugging Distributed Systems

CHALLENGES AND OPTIONS FOR VALIDATION AND DEBUGGING

IVAN BESCHASTNIKH

PATTY WANG

YURIY BRUN

MICHAEL D. ERNST

Distributed systems pose unique challenges for software developers. Reasoning about concurrent activities of system nodes and even understanding the system's communication topology can be difficult. A standard approach to gaining insight into system activity is to analyze system logs. Unfortunately, this can be a tedious and complex process. This article looks at several key features and debugging challenges that differentiate distributed systems from other kinds of software. The article presents several promising tools and ongoing research to help resolve these challenges.

DISTRIBUTED-SYSTEM FEATURES AND CHALLENGES

Distributed systems differ from single-machine programs in ways that are simultaneously positive in providing systems with special capabilities, and negative in presenting software-development and operational challenges.

Heterogeneity

A distributed system's nodes may include mobile phones, laptops, server-class machines, and more. This hardware and software diversity in node resources and network connectivity can make a distributed system more robust, but this heterogeneity forces developers to manage compatibility during both development and debugging.

Concurrency

Simultaneous operation by multiple nodes leads to concurrency, which can make a distributed system outperform a centralized system. However, concurrency may introduce race conditions and deadlocks, which are notoriously difficult to diagnose and debug. Additionally, networks introduce packet delay and loss, exacerbating the issues of understanding and debugging concurrency.

Distributed state

Distributing system state across multiple nodes can remove a central point of failure and improve scalability, but distributed state requires intricate node coordination to synchronize state across nodes—for example, nodes must ensure their local states are consistent. Potential inconsistencies are prevented by distributed algorithms, such as those that guarantee a particular flavor of data consistency and cache coherence. Developers may find it difficult, or even impossible, to reconstruct the global state of the system when it is distributed on many nodes. This complicates bug diagnosis and validation.

Partial failures

The distribution of state and responsibility allows distributed systems to be robust and survive a variety of failures. For example, Google's Spanner system can survive failures of

For most distributed systems, fault tolerance cannot be an afterthought; the systems must be designed to deal with failures.

entire data centers.[2] Achieving such fault tolerance, however, requires developers to reason through complex failure modes. For most distributed systems, fault tolerance cannot be an afterthought; the systems must be designed to deal with failures. Such failure resiliency is complex to design and difficult to test.

EXISTING APPROACHES

What follows is an overview of seven approaches designed to help software engineers validate and debug distributed systems.

Testing

A test suite exercises a specific set of executions to ensure that they behave properly. Most testing of distributed systems is done using manually written tests, typically introduced in response to failures and then minimized.[13] Testing is an effective way to detect errors. However, since testing exercises a limited number of executions, it can never guarantee to reveal all errors.

Model checking

Model checking is exhaustive testing, typically up to a certain bound (number of messages or steps in an execution). Symbolic model checking represents and explores possible executions mathematically; explicit-state model checking is more practical because it actually runs the program, controlling its executions rather than attempting to abstract it. MoDist performs black-box model checking, permuting message sequences and changing the execution speed of a process relative to other processes in the system.[17] MaceMC is a white-box technique that achieves speedups by adding programming-language support for model checking.[6] Common problems of all model-checking tools are scalability and environmental modeling, so they rarely achieve a guarantee.

Theorem proving

Theorem proving can, in principle, prove a distributed system to be free of defects. Amazon uses TLA+ to verify its distributed systems.[10] Two recent systems can construct a verified distributed-system implementation. Verdi uses the Coq tool, whose expressive type system makes type checking equivalent to theorem proving, thanks to the Curry-Howard isomorphism; the Coq specification is then compiled into an OCaml implementation of the distributed system.[15] In contrast, IronFleet uses TLA and Hoare-logic verification to similarly produce a verified implementation of a distributed system.[5] The enormous effort needed to use these tools makes them most appropriate for new implementations of small, critical cores. Other techniques are needed for existing distributed systems.

Record and replay

Record and replay captures a single execution of the system so that this execution can be later replayed or analyzed. This is especially useful when debugging nondeterministic behaviors. A record-and-replay tool such as Friday[4] or

D3S[7] captures all nondeterministic events so that an execution can be reproduced exactly. Recording a complex execution, however, may be prohibitively expensive and may change the behavior of the underlying system.

Tracing

Tracing tracks the flow of data through a system, even across applications and protocols such as a database, web server, domain-name server, load balancer, or virtual private network protocol.[12] For example, pivot tracing dynamically instruments Java-based systems to collect user-defined metrics at different points in the system and collates the resulting data to provide an inter-component view of the metrics over multiple executions.[8] Dapper is a lower-level tracing system used at Google to trace infrastructure services.[14] Tracing is more efficient than record and replay because it focuses on a specific subset of the data, but it requires instrumenting applications and protocols to properly forward, without consuming, the tracing metadata.

Log analysis

Log analysis is an even lighter-weight approach that works with systems that cannot be modified. It is a common black-box approach in which a system's console logs, debug logs, and other log sources are used to understand the system. For example, Xu et al. applied machine learning to logs to detect anomalies in Google infrastructure services.[16] Detailed logs from realistic systems contain a great deal of valuable detail, but they tend to be so large that they are overwhelming to programmers, who as a result cannot directly benefit from them.

Visualization

The complexity of distributed systems has inspired work on visualization of such systems to make them more transparent to developers. For example, Theia displays a visual signature that summarizes various aspects of a Hadoop execution, such as the execution's resource utilization.[3] These signatures can be used to spot anomalies and to compare executions. Tools such as Theia provide high-level summaries of a system's behavior. They do not, however, help a developer understand the underlying communication pattern in the system, including the distributed ordering of messages.

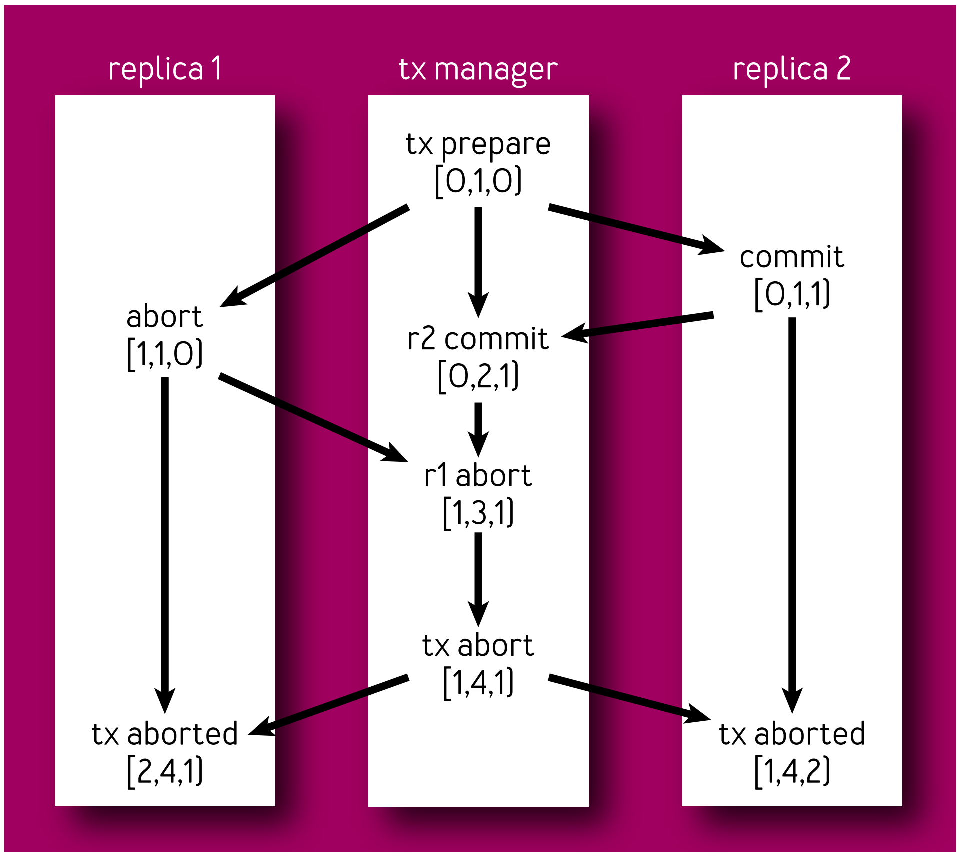
VISUALIZING DISTRIBUTED-SYSTEM EXECUTIONS

As noted above, the ability to visualize distributed-system executions can help developers understand and debug their distributed systems. ShiViz is such a visualization tool, displaying distributed-system executions as interactive time-space diagrams that explicitly capture distributed ordering of messages and events in the system. This diagram reproduces the events and interactions captured in the execution log, making the ordering information explicit through a concise visualization. A developer can expand, collapse, and hide parts of the diagram, as well as search for particular interaction patterns. ShiViz is freely available as a browser application; any developer can visualize a log, without installing software or sending the log over the network.

To provide a rich and accurate visualization of a distributed system's execution, ShiViz displays the *happens-before relation*. Given event *e* at node *n*, the happens-before relation indicates all the events that logically precede *e*. Other events might have already occurred at other nodes according to wall-clock time, but node *n* cannot tell whether those other events happened before or after *e*, and they do not affect the behavior of *e*. This partial order can rule out which events do not cause others, identify concurrent events, and help developers mentally replay parts of the execution.

Figure 1 illustrates an execution of the two-phase commit protocol with one transaction manager and two replicas.[1]

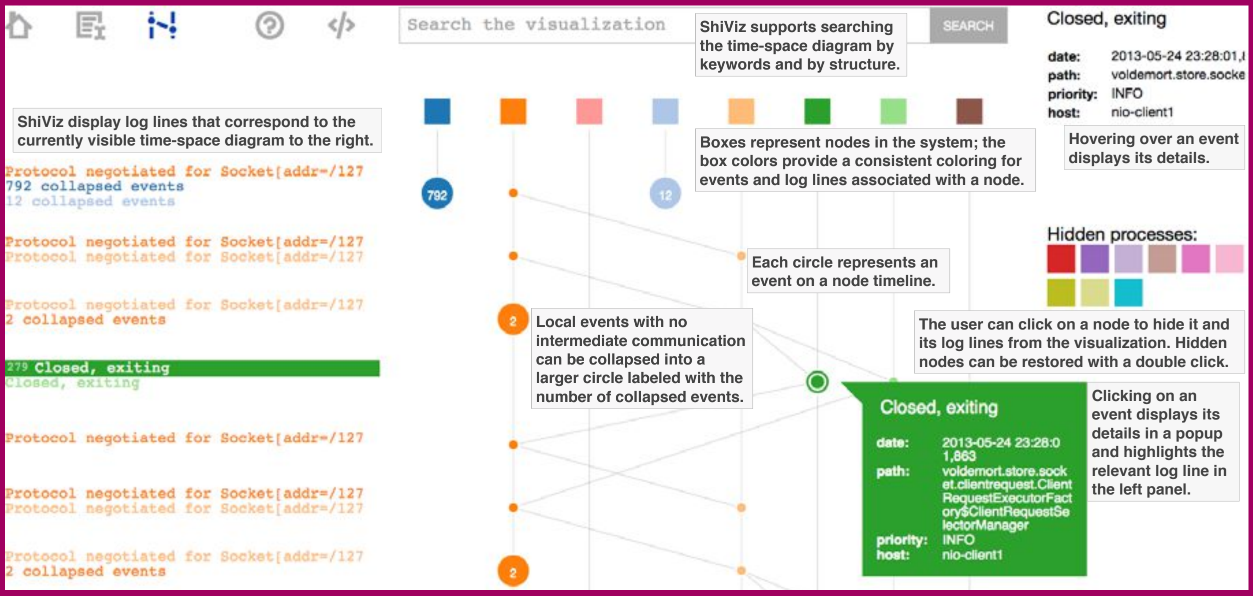
FIGURE 1: **TIME-SPACE DIAGRAM OF AN EXECUTION WITH THREE NODES**



This time-space diagram is a visualization of the underlying happens-before partial order, showing an execution with three nodes. Lines with arrows denote the partial ordering of events, each of which has an associated vector timestamp in brackets. (See timestamp sidebar on next page.)

Figure 2 shows a screenshot of ShiViz visualizing an execution of a distributed data-store system called Voldemort.[11] In the middle of the screen is the time-space diagram, with time flowing from top to bottom. The colored boxes at the top represent nodes, and the vertical lines below them are the node timelines. Circles on each node's timeline represent events executed by that node. 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

FIGURE 2: **A SHIVIZ SCREENSHOT**



ShiViz supports searching the time-space diagram by keywords and by structure.

ShiViz display log lines that correspond to the currently visible time-space diagram to the right.

```
Protocol negotiated for Socket[addr=/127
792 collapsed events
12 collapsed events

Protocol negotiated for Socket[addr=/127
Protocol negotiated for Socket[addr=/127

Protocol negotiated for Socket[addr=/127
2 collapsed events

279 Closed, exiting
Closed, exiting

Protocol negotiated for Socket[addr=/127

Protocol negotiated for Socket[addr=/127
Protocol negotiated for Socket[addr=/127

Protocol negotiated for Socket[addr=/127
2 collapsed events
```

Boxes represent nodes in the system; the box colors provide a consistent coloring for events and log lines associated with a node.

Each circle represents an event on a node timeline.

Local events with no intermediate communication can be collapsed into a larger circle labeled with the number of collapsed events.

Closed, exiting

date:	2013-05-24 23:28:01,(
path:	voldemort.store.socke
priority:	INFO
host:	nio-client1

Hovering over an event displays its details.

Hidden processes:

The user can click on a node to hide it and its log lines from the visualization. Hidden nodes can be restored with a double click.

Closed, exiting

date:	2013-05-24 23:28:01,863
path:	voldemort.store.socket.clientrequest.ClientRequestExecutorFactory$ClientRequestSelectorManager
priority:	INFO
host:	nio-client1

Clicking on an event displays its details in a popup and highlights the relevant log line in the left panel.

to via a downward path. ShiViz augments the time-space diagram with operations to help developers explore distributed-system executions and corresponding logs. Figure 2 details some of these operations.

Distributed timestamps

A typical distributed-system log does not contain enough information to regenerate the happens-before relation, and this is one reason that distributed-system logs are so hard to interpret. ShiViz relies on logs that have been enhanced by another tool, ShiVector, to include *vector clock timestamps* that capture the happens-before relation between events.[9] Each node α maintains a vector of logical clocks, one clock for each node in the distributed system, including itself. α's *i*th clock is a lower bound on the current logical time at node *i*. The node α increments the αth component of its vector clock each time it performs a local action or sends or receives a message. Each message contains the sending node's current vector clock; upon message receipt, the receiving node updates its vector clock to the elementwise maximum of its local and received timestamps.

ShiVector is a lightweight instrumentation tool that augments the information already logged by a distributed system with the partial ordering information encoded as vector clocks. ShiVector interposes on communication and logging channels at each node in the system to add vector clock timestamps to every logged event.

ShiViz parses ShiVector-augmented logs to determine, for each event: (1) the node that executed the event; (2) the vector timestamp of the event; and (3) the event's description.

ShiViz permits a user to customize the parsing of logs using regular expressions, which can be used to associate additional information, or *fields*, with each event.

UNDERSTANDING DISTRIBUTED-SYSTEM EXECUTIONS

ShiViz helps developers (1) to understand the relative ordering of events and the likely chains of causality between events, which is important for debugging concurrent behavior; (2) to query for certain events and interaction patterns between hosts; and (3) to identify structural similarities and differences between pairs and groups of executions. The time-space diagram representation supports the first goal by visualizing event ordering and communication. The next section describes two search operations that support the second goal, and operations over multiple executions that correspond to the third goal.

Keyword search and structured search operations

ShiViz implements two kinds of search operations: keyword and structured. Both types are accessible to the developer through the top search bar (see figure 2).

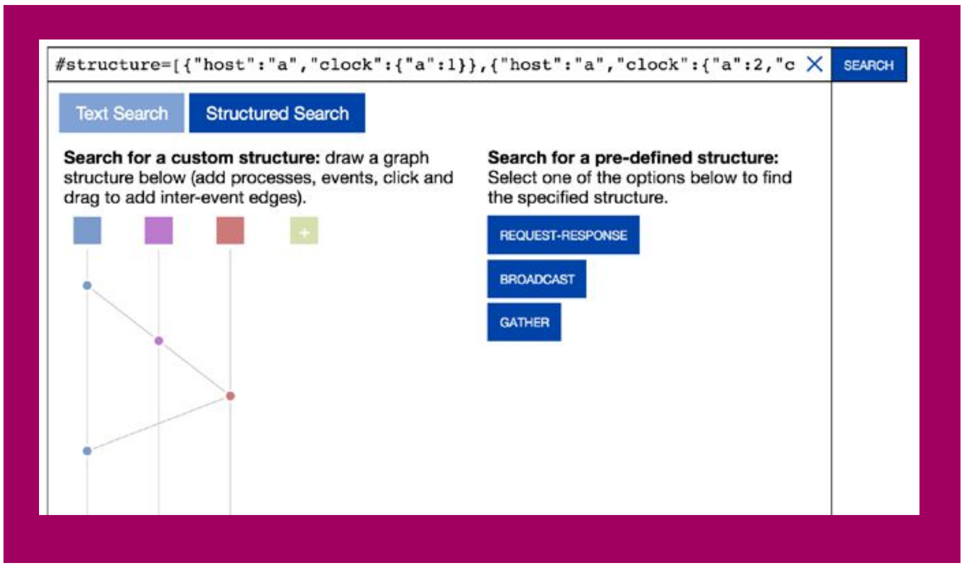
Keyword search allows a developer to highlight all events in the diagram that contain a field matching a query. For example, searching for `send` will highlight all events in the diagram that have a field whose value is `send`. The results can be further constrained with field identifiers and regular expressions. For example, the query `node=alice && priority=CRITICAL*` will highlight only events at the `alice` node with a priority field matching the regular expression `CRITICAL*`.

In a structured search, a user queries ShiViz for any set of events related through a particular ordering pattern, and ShiViz highlights the sections of the diagram (events and their interconnections) that match this pattern. ShiViz includes several predefined patterns:

- Request-response. A source node sends a request and the destination node sends back a response.
- Broadcast. A node sends a message to most other nodes in the system.
- Gather. A node receives a message from most other nodes.

A user can also compose a custom pattern consisting of nodes, node events, and connections between events representing a partial order. Figure 3 shows such a custom pattern, depicting three nodes communicating in a ring: node 1 communicates only with node 2; node 2 with node 3; and node 3 with node 1. Drawing this pattern allows the user to search for all instances of this three-node ring communication in the execution. ShiViz automatically translates the drawn pattern into a textual representation (see search bar at the top), and it is possible to edit,

FIGURE 3: **STRUCTURED SEARCH FEATURE**



```
#structure=[{"host":"a","clock":{"a":1}},{"host":"a","clock":{"a":2,"c
```

Search for a custom structure: draw a graph structure below (add processes, events, click and drag to add inter-event edges).

Search for a pre-defined structure: Select one of the options below to find the specified structure.

copy, and paste the textual representation directly. The structured search feature allows users to express custom communication patterns between events and to query an execution for instances of the specified pattern. The presence or absence of queried subgraphs at particular points in an execution can help users detect anomalous behavior, aiding them in their debugging efforts.

Comparing executions

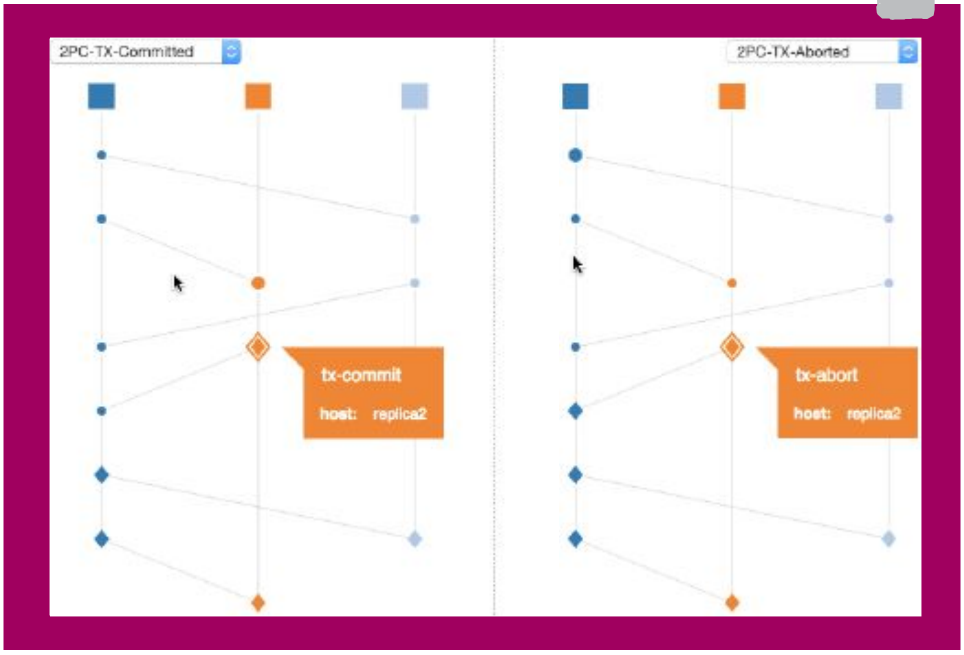
ShiViz can help users understand multiple executions of a system. When ShiViz parses multiple executions, the user can choose between viewing executions individually or pairwise.

In the pairwise view, a user can compare the two executions further by highlighting their differences. When enabled, the nodes are compared by name. For nodes present in both executions, ShiViz compares their events one by one by comparing the corresponding event descriptions. Nodes or events in one execution that do not appear in the other are redrawn as rhombuses.

Figure 4 illustrates this pairwise comparison on a log of the two-phase commit protocol. The two selected events in the figure explain the difference between these two executions: the two-phase commit successfully commits a transaction in the left execution, but aborts a transaction in the right execution.

The explicit highlighting of differences provides users with fast detection of anomalous events or points where

FIGURE 4: **TWO TWO-PHASE COMMIT PROTOCOL EXECUTIONS**



the two executions diverge. The search features described earlier can be applied in the pairwise view to help developers detect specific unifying or distinguishing features across traces, allowing them to design and test their systems more effectively.

Clustering executions

To help manage many executions, ShiViz supports grouping executions into clusters. A user can cluster by the number of nodes or by comparison to a base execution, using as a distance metric the differencing mechanism described earlier. 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. Users 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 users pinpoint executions of interest by allowing them to inspect a subset of executions matching a desired measure. This subset can be further narrowed by performing a keyword search or a structured search on top of the clustering results. Execution names among clusters are highlighted if their corresponding graphs contain instances matching the user's search query.

ShiViz helps developers visualize the event order, search for communication patterns, and identify potential event causality. This can help developers reason about the concurrency of events in an execution's distributed system state, and distributed failure modes, as well as formulate hypotheses about system behavior and verify them via execution visualizations. Meanwhile, the generality of logging makes ShiVector and ShiViz broadly applicable to systems deployed on a wide range of devices.

ShiViz has some limitations. ShiViz surfaces low-level ordering information, which makes it a poor choice for understanding high-level system behavior. The ShiViz visualization is based on logical and not realtime 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. This design choice, however, also limits its scalability.

ShiViz is an open-source tool with an online deployment (http://bestchai.bitbucket.org/shiviz/). Watch a video demonstrating key ShiViz features at http://bestchai.bitbucket.org/shiviz-demo/.

Acknowledgments

We thank Perry Liu and Albert Xing, who helped develop ShiViz; Jenny Abrahamson, who developed the initial ShiVector and ShiViz prototypes; and Donald Acton and Colin Scott, who helped evaluate ShiViz. This work is supported by NSERC USRA, the NSERC Discovery grant, and the National Science Foundation under grants CCF-1453474 and CNS-1513055. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0107. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes, notwithstanding any copyright notices thereon.

References

1. Bernstein, P., Hadzilacos, V., Goodman, N. 1986. Distributed recovery. In *Concurrency Control and Recovery in Database Systems*, Chapter 7. Addison-Wesley; http://research.microsoft.com/en-us/people/philbe/chapter7.pdf.
2. Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D. 2012. Spanner: Google's globally distributed database. 10th Usenix Symposium on Operating Systems

Design and Implementation; https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett.

3. Garduno, E., Kavulya, S. P., Tan, J., Gandhi, R., Narasimhan, P. 2012. Theia: visual signatures for problem diagnosis in large Hadoop clusters. *Proceedings of the 26th International Conference on Large Installation System Administration*: 33-42; https://users.ece.cmu.edu/~spertet/papers/hadoopvis-lisa12-cameraready-v3.pdf.
4. Geels, D., Altekar, G., Maniatis, P., Roscoe, T., Stoica, I. 2007. Friday: global comprehension for distributed replay. *Proceedings of the Fourth Usenix Conference on Networked Systems Design and Implementation*; https://www.usenix.org/legacy/event/nsdi07/tech/full_papers/geels/geels.pdf.
5. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., Zill, B. 2015. IronFleet: proving practical distributed systems correct. *Proceedings of the 25th Symposium on Operating Systems Principles*; http://sigops.org/sosp/sosp15/current/2015-Monterey/250-hawblitzel-online.pdf.
6. Killian, C., Anderson, J. W., Jhala, R., Vahdat, A. 2007. Life, death, and the critical transition: finding liveness bugs in systems code. *Proceedings of the Fourth Usenix Conference on Networked Systems Design and Implementation*; https://www.usenix.org/legacy/event/nsdi07/tech/killian/killian.pdf.
7. Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu,

M., Kaashoek, M. F., Zhang, Z. 2008. D3S: debugging deployed distributed systems. *Proceedings of the Fifth Usenix Symposium on Networked Systems Design and Implementation*: 423-437; http://static.usenix.org/event/nsdi08/tech/full_papers/liu_xuezheng/liu_xuezheng.pdf.

8. Mace, J., Roelke, R., Fonseca, R. 2015. Pivot tracing: dynamic causal monitoring for distributed systems. *Proceedings of the 25th Symposium on Operating Systems Principles*: 378-393; http://sigops.org/sosp/sosp15/current/2015-Monterey/122-mace-online.pdf.
9. Mattern, F. 1989. Virtual time and global states of distributed systems. *Proceedings of the International Workshop on Parallel and Distributed Algorithms*; http://homes.cs.washington.edu/~arvind/cs425/doc/mattern89virtual.pdf.
10. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M. 2015. How Amazon Web Services uses formal methods. *Communications of the ACM* 58(4): 66-73; http://cacm.acm.org/magazines/2015/4/184701-how-amazon-web-services-uses-formal-methods/fulltext.
11. Project Voldemort; http://www.project-voldemort.com/voldemort/.
12. Sambasivan, R. R., Fonseca, R., Shafer, I., Ganger, G. 2014. So, you want to trace your distributed system? Key design insights from years of practical experience. Parallel Data Laboratory, Carnegie Mellon University; http://www.pdl.cmu.edu/PDL-FTP/SelfStar/CMU-PDL-14-102.pdf.
13. Scott, C., Wundsam, A., Raghavan, B., Panda, A., Or, A., Lai, J., Huang, E., Liu, Z., El-Hassany, A., Whitlock, S., Acharya,

H. B., Zarifis, K., Shenker, S. 2014. Troubleshooting blackbox SDN control software with minimal causal sequences. *Proceedings of the ACM Conference on SIGCOMM*: 395-406; https://www.eecs.berkeley.edu/~apanda/papers/sts.pdf.

14. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C. 2010. Dapper, a large-scale distributed systems tracing infrastructure. Research at Google; http://research.google.com/pubs/pub36356.html.
15. Wilcox, J. R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M. D., Anderson, T. 2015. Verdi: a framework for implementing and formally verifying distributed systems. *Proceedings of the 36th SIGPLAN Conference on Programming Language Design and Implementation*: 357-368; https://homes.cs.washington.edu/~ztatlock/pubs/verdi-wilcox-pldi15.pdf.
16. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M. 2010. Experience mining Google's production console logs. *Proceedings of the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*; http://iiis.tsinghua.edu.cn/~weixu/files/slaml10.pdf.
17. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L. 2009. MoDist: transparent model checking of unmodified distributed systems. *Proceedings of the Sixth Usenix Symposium on Networked Systems Design and Implementation*: 213-228; https://www.usenix.org/legacy/event/nsdi09/tech/full_papers/yang/yang_html/.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Ivan Beschastnikh *works on improving the design, implementation, and operation of complex systems. He is an assistant professor in the department of computer science at the University of British Columbia, where he leads a team of students on projects that span distributed systems, software engineering, security, and networks, with a particular focus on program analysis. More information is available at his homepage: http://www.cs.ubc.ca/~bestchai/.*

Patty Wang *earned her bachelor's degree in computer science and mathematics from the University of British Columbia. She has explored approaches to helping developers understand and compare multiple distributed executions, focusing on summarizing similarities and differences across traces.*

Yuriy Brun *works on automating system building and creating self-adaptive systems. He is an assistant professor at the University of Massachusetts, Amherst. He received his PhD from the University of Southern California in 2008, and then spent three years as a postdoctoral fellow at the University of Washington. He has been recognized with a CAREER award from the National Science Foundation, a Microsoft Research SEIF (Software Engineering Innovation Foundation) award, a Google Faculty Research award, and an IEEE TCSC (Technical Committee on Scalable Computing) Young Achiever in Scalable Computing award. More information is available at his homepage: http://people.cs.umass.edu/~brun/.*

Michael D. Ernst *researches ways to make software more reliable, more secure, and easier (and more fun!) to produce. His primary technical interests are in software engineering, programming languages, type theory, security, program analysis, bug prediction, testing, and verification. Ernst is an ACM Fellow and received the inaugural John Backus Award, the NSF CAREER Award, 10 best-paper awards, and other honors. More information is available at his homepage: http://homes.cs.washington.edu/~mernst/.*

CONTENTS