Visualizing Request-Flow Comparison to Aid Performance Diagnosis in Distributed Systems

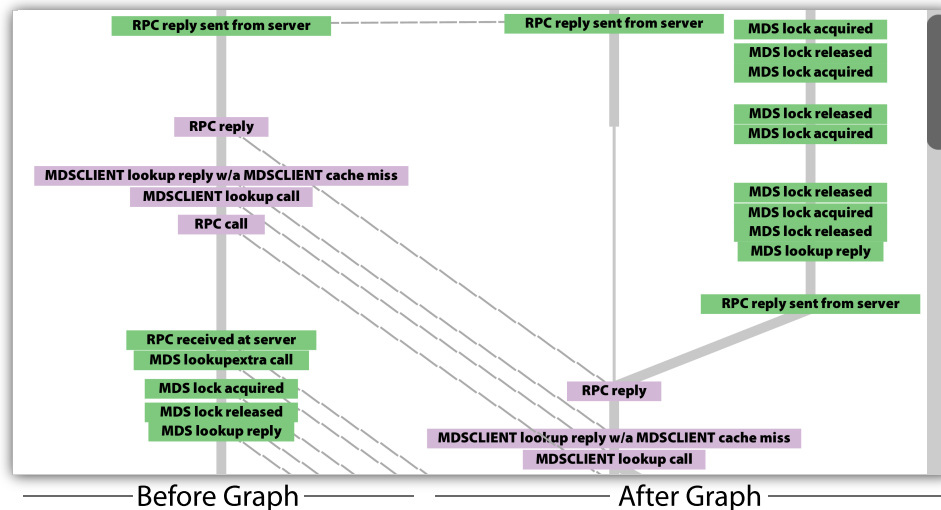Raja R. Sambasivan, Ilari Shafer, Michelle L. Mazurek, and Gregory R. Ganger

Fig. 1. **Comparing request-flow graphs:** This side-by-side visualization, one of three interfaces we evaluate, illustrates the output of a diagnosis technique that compares graphs. It shows these two graphs juxtaposed horizontally, with dashed lines between matching nodes in both. The rightmost series of nodes in the screenshot do not exist in the graph on the left, causing the yellow nodes to shift downward in the graph on the right.

Abstract—Distributed systems are complex to develop and administer, and performance problem diagnosis is particularly challenging. When performance degrades, the problem might be in any of the system's many components or could be a result of poor interactions among them. Recent research efforts have created tools that automatically localize the problem to a small number of potential culprits, but research is needed to understand what visualization techniques work best for helping distributed systems developers understand and explore their results. This paper compares the relative merits of three well-known visualization approaches (side-by-side, diff, and animation) in the context of presenting the results of one proven automated localization technique called *request-flow comparison*. Via a 26-person user study, which included real distributed systems developers, we identify the unique benefits that each approach provides for different problem types and usage modes.

Index Terms—Distributed systems, human factors, problem diagnosis, visualization

✦

1 INTRODUCTION

A distributed system is a set of software components running on multiple networked computers that collectively provide some service or result. Examples now pervade all walks of life, as society uses distributed services to communicate (e.g., Google's Gmail), shop (e.g., Amazon), provide entertainment (e.g., YouTube), and so forth. Though such distributed systems often have simple interfaces and usually respond quickly, great complexity is involved in developing them and maintaining their performance levels over time. Unexpected performance degradations arise frequently, and substantial human effort is involved in addressing them.

When a performance degradation arises, the crucial first step in addressing it is figuring out what is causing it. The *root cause* might be any of the system's software components, unexpected interactions

---

- *Raja R. Sambasivan is with Carnegie Mellon. E-mail: rajas@ece.cmu.edu.*
- *Ilari Shafer is with Carnegie Mellon. E-mail: ishafer@cs.cmu.edu.*
- *Michelle L. Mazurek is with Carnegie Mellon. E-mail: mmazurek@cmu.edu.*
- *Gregory R. Ganger is with Carnegie Mellon. E-mail: ganger@ece.cmu.edu.*

between them, or slowdowns in the network connecting them. Exploring the possibilities and identifying the most likely root causes has traditionally been an ad-hoc manual process, informed primarily by raw performance data collected from individual components. As distributed systems have grown in scale and complexity, such ad-hoc processes have grown less and less tenable.

To help, recent research has proposed many techniques for automatically localizing the many possible sources of a new problem to just a few potential culprits [21, 27, 32]. These techniques do not identify the root cause directly, but rather help developers build intuition about the problem and focus their diagnosis efforts. Though complete automation would be ideal, the complexity of modern systems and the problems that arise in them ensure this human-in-the-loop model will be dominant for the foreseeable future. As such, many researchers recognize the need for diagnosis tools to present their results as clearly as possible [24, 28]. But, apart from a few select instances [22, 24], far too little research has been conducted on what types of presentations are most useful for distributed systems developers.

As a step toward addressing this need, this paper presents a 26-person user study that compares three promising approaches for visualizing the results of an implementation of one powerful, proven automated localization technique called *request-flow comparison* [32]. The approaches we consider are well-known in the visualization community, and we make no claims as to their novelty. Rather, our contribution lies in identifying which of these approaches work best for diagnosing

---

different types of distributed systems problems and for different developer usage modes. Our user study uses real problems observed in Ursa Minor [1], a real distributed storage system. It includes 13 professionals (i.e., developers of Ursa Minor and software engineers from Google) and 13 graduate students taking distributed systems classes.

Request-flow comparison contrasts how a distributed system services requests (e.g., "read this e-mail message" or "find books by this author") during two periods of operation: one where performance was fine ("before") and the new one in which performance has degraded ("after"). Each serviced request has a corresponding workflow within the system, representing the order and timing of components involved; for example, a request to read e-mail might start at a frontend web server that parses the request, then be forwarded to the e-mail directory server for the specific user, then be forwarded to the storage server that holds the desired message, and then return to the web server so it can respond to the requester. Figure 2 shows a similar example for a generic distributed storage system. Each such request flow can be represented as a directed acyclic graph, and comparing before and after graphs can provide significant insight into performance degradations. Many organizations are interested in algorithms and visualizations for comparing request flows, including Google [35], Microsoft, and Twitter [37].

The initial version of our request-flow-comparison implementation, Spectroscope [32], used an inadequate Graphviz-based interface that required graphs to be manually and painstakingly compared with each other. The approaches we compare in this paper were chosen based on the recommendations of developers who used this initial version to diagnose real problems in Ursa Minor and certain Google services [32]. They occupy three corners in the space of approaches to visualizing differences, as identified by a taxonomy of comparison approaches [17]. The side-by-side approach is nearly a "juxtaposition," which presents independent layouts. Diff is an "explicit encoding," which highlights the differences between the two graphs. Animation is closest to a "superposition" design that guides attention to changes that "blink."

Despite the large body of work on comparing graphs [4, 5, 13], we found no existing implementations of side-by-side, diff, and animation that are suitable for request-flow comparison's domain-specific needs. For example, differences must be found in directed acyclic graph structure and edge weights. Also, matching nodes in before-and-after graphs are not known a priori and must be identified algorithmically. Therefore, we built our own interfaces.

Our user study results show that side-by-side is the best approach for helping developers obtain an overall understanding of a problem. Diff is best for helping diagnose problems caused by non-structural differences. Animation is best for helping diagnose problems that are caused by structural differences alone.

2 REQUEST-FLOW COMPARISON

Request-flow comparison [32] is a technique for automatically localizing the root causes of performance degradations in distributed systems, such as Ursa Minor and Bigtable [9]. Such degradations often manifest as changes in the workflow of individual requests as they are serviced by the system. Exposing such changes and showing how they differ from previous behavior localizes the problem's source and significantly guides developer effort.

Request-flow comparison aims to identify changes between request-flow graphs observed during two periods: one obtained before the degradation and one obtained after the degradation. Nodes of request-flow graphs (which are directed acyclic graphs) show important events or activities observed within different components during request processing, and edges show latencies between these events (see Figure 3). Request-flow comparison identifies two important types of changes. *Edge latency changes* are differences in the time required to execute successive events and represent unforeseen slowdowns in request processing. *Structural changes* are differences in the number and causal ordering of system events. They represent slowdowns due to extra (or less) concurrent activity or additional (or different) activity within a single sequential thread (i.e., intra-thread event changes).

Spectroscope [32], our current prototype, implements request-flow comparison by grouping identically-structured graphs observed during
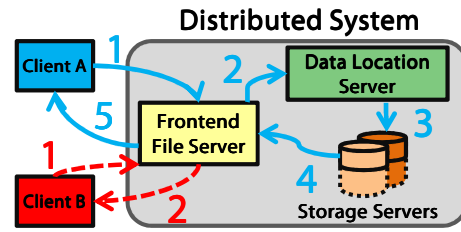

Fig. 2. **A distributed storage system.** To read a file, clients send a request to the frontend file server. To return the requested data, the file server may need to access other components like the data location server and storage servers. This figure shows two of many possible flows through the system.

both periods (often numbered in the hundreds or thousands) into the same *category*. By analyzing per-period statistical properties, Spectroscope identifies categories containing performance-affecting changes. As output, Spectroscope presents *before-and-after* pairs of these categories showing how request processing changed between the two periods. Since categories contain identically structured requests, each is represented by a single constituent request-flow graph. Edge latency changes are automatically identified via hypothesis testing, but such tests may not identify all edges worth investigating, so developers must still examine before-after graph pairs manually to find additional such divergences. Similar tests are used in several automated diagnosis tools [21, 27, 32]. Spectroscope will identify categories containing structural changes, but developers must contrast before-after graphs manually to identify where the divergences occur.

3 RELATED WORK

Recent research has explored a number of approaches, including some akin to side-by-side, diff, and animation, to help users identify differences in graphs, but no single one has emerged as the clear winner [4, 5, 31]. The choice depends on the domain, the types of graphs being compared, and the differences of interest, thus motivating the need for the study presented in this paper. This section discusses user studies comparing different approaches for comparing graphs, as well as tools designed to identify graph differences.

For understanding the evolution of undirected, unweighted graphs (e.g., many social network graphs) over time, Archambault et al. [4] find that small multiples (akin to our side-by-side approach) yielded faster task completion times, but that accuracy was greater with animation for tasks that involved identifying node or edge additions/deletions.
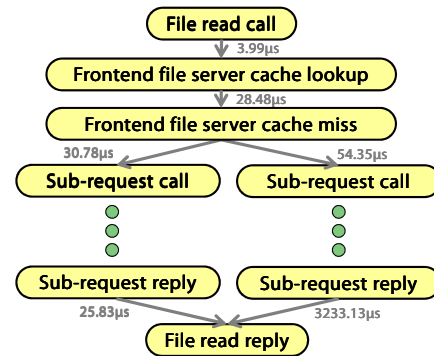

Fig. 3. **Example request-flow graph.** This graph shows the flow of a read request through the distributed storage system shown in Figure 2. Node names represent important events observed on the various components while completing the required work. Edges show latencies between these events. Fan-outs represent the start of parallel activity, and synchronization points (i.e., joins) are indicated by fan-ins. Due to space constraints, only the events observed on the frontend file server are shown.

In contrast, Farrugia et al. [13] find that small multiples outperform animation with regard to both completion time and accuracy. In a later study, Archambault et al. [5] evaluate a hybrid approach that uses side-by-side or animation to present diffs of consecutive graph evolutions. Participants preferred the hybrid approach over comparing independent graphs of each evolutionary stage, but, it did not yield lower completion times or increased accuracy for most tasks. Melville et al. [25] develop a set of general graph-comparison questions and find that for small graphs represented as adjacency matrices, a superimposed (diff) view is superior to a juxtaposed (side-by-side) view.

For weighted graphs that are much denser than our request-flow graphs, Alper et al. [2] find that matrix-based superimposed representations outperform node-link-based ones. Robertson et al. [31] compare animation's effectiveness to that of small multiples and one other static approach for identifying trends in the evolution of Gapminder Trendalyzer [15] plots (3-dimensional data plotted on 2-D axes). They find that animation is more effective for presenting trends, while static approaches are more effective for helping users identify them.

Many tools designed to identify graph differences use domain-specific algorithms or target specific graph structures. For example, TreeJuxtaposer [26] uses domain knowledge to match similar nodes in trees that show evolutionary relationships among different species. TreeVersity [18] uses a diff-based technique to identify differences in node attributes and structure for trees with unweighted edges and known matchings. G-PARE [34] presents overlays to compare predictions made by machine-learning algorithms on graph-based data. Visual Pivot [30] helps identify relationships between two trees by using a layout that co-locates a selected common node. Beck and Diehl [6] use a matrix representation to compare software architectures based on code dependencies. Donatien [19] enables comparison of unweighted graphs by placing individual graphs in independently-configurable overlays, which can be optionally superimposed or viewed side-by-side. Their human-in-the loop model for graph matching could be applied to our interfaces to improve matching accuracy.

Our work in this paper complements both of the above related-work categories. Specifically, we attempt to identify good graph comparison techniques for the output of a tool used in one important domain: performance diagnosis of distributed systems using request-flow comparison. Our domain is characterized by sparse, directed, acyclic, weighted graphs, with unknown matchings. We also believe our intended audience—those familiar with distributed systems development—will exhibit different preferences than the general population.

In the systems community, there has been relatively little research conducted on visual methods for diagnosis. Indeed, a recent survey of important directions for log analysis concludes that because humans will remain in the diagnosis loop for the foreseeable feature, visualization research is an important next step [28]. One project in this vein is NetClinic, which considers diagnosis of network faults [22]. The authors find that visualization in conjunction with automated analysis [21] is helpful for diagnosis. As in this study, their tool uses automated processes to direct users' attention, and the authors observe that automation failures inhibit users' understanding. In another system targeting network diagnosis, Mansmann et al. observe that automated tools are limited in utility without effective presentation of results [24].

4 INTERFACE DESIGN

Figure 4 shows our implementations of side-by-side, diff, and animation. All of them were guided by an initial pilot study, which is not described in this paper. These interfaces are implemented in JavaScript, and use modified libraries from the JavaScript InfoVis Toolkit [7]. This section further describes them.

Side-by-side: Our side-by-side interface (illustrated with a simplified diagram at right and in Figure 4a,d) computes independent layered layouts for the before and after graphs and displays them beside each other. Nodes in the before graph are linked to matching nodes in the after
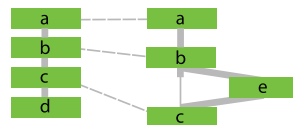

graph by dashed *correspondence lines*. Using this interface, latency changes can be identified by examining the relative slope of adjacent dashed lines: parallel lines indicate no change in latency, while increasing skew is indicative of longer response time. Structural changes can be identified by the presence of nodes in the before or after graph with no matching node in the other graph.

Diff: Our diff interface (at right and in Figure 4b,e) shows a single static image of an explicit encoding of the differences between the before and after graphs, which are associated with the colors orange and blue respectively. The layout contains all nodes from both the before and after graphs. Nodes that exist only in the before graph are outlined in orange and annotated with a minus sign; those that exist

only in the after graph are outlined in blue and annotated with a plus sign. This structural approach is akin to the output of a contextual diff tool [23] emphasizing insertions and deletions.
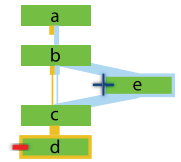
We use the same orange and blue scheme to show latency changes, with edges that exist in only one graph shown in the appropriate color. Edges existing in both graphs produce a per-edge latency diff: orange and blue lines are inset together with different lengths. The ratio of the lengths is computed from the ratio of the edge latencies in before and after graphs, and the next node is attached at the end of the longer line.

Animation: Our animation interface (at right and in Figure 4c,f) switches automatically between the before and after graphs. Independent layouts are calculated for each graph, but non-matching nodes are not allowed to occupy the same position. To provide a smooth transition, it interpolates the positions of matching nodes. Nodes that exist in only one graph appear only on the

appropriate terminal of the animation. They all become more transparent at the same rate, and vanish completely by the other terminal. Both types of transition are linear and happen concurrently.
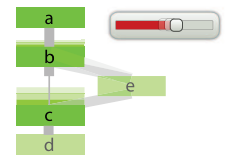
Users can start and stop the animation, as well as manually switch between terminal or intermediate points, via the provided slider. Unless it is controlled this way, the animation repeats with a transition period of two seconds and a pause for three seconds at either terminal.

4.1 Graph matching

Drawing correspondence lines in the side-by-side interface and merging corresponding nodes in the diff and animation interfaces requires identifying matching nodes in the before and after graphs. This problem is hard (in the formal sense), and there exists a wide body of work investigating algorithms and heuristics for such matching [8, 11, 14]. Like many of these approaches, we use heuristics to obtain approximate matches.

Our approximation technique, which runs in $O(N^2)$ time in the number of nodes, makes use of the fact that if a node in the before graph matches a node in the after graph, their node names will be the same. (Note the converse is not necessarily true.) First, we use a lexically-ordered depth-first traversal to transform both graphs into strings. Next, we keep track of the insertions, deletions, and mutations made by a string-edit distance transformation of one string into another. Finally, we map these edits back onto the appropriate interface. Items that were not inserted, deleted, or modified are ones that correspond in both graphs. A given node in the before graph is matched to at most one node in the after graph. Despite the limitations of this approach, we have found it to work well in practice.

4.2 Common features

All three of our interfaces incorporate some common features, tailored specifically for request-flow graphs. As seen above, all graphs are represented with node-link diagrams since request-flow graphs are sparse, directed graphs that have fan-ins and fan-outs. They are drawn with a layered layout based on the technique by Sugiyama et al. [36]; layouts based on this technique enjoy widespread use [12].
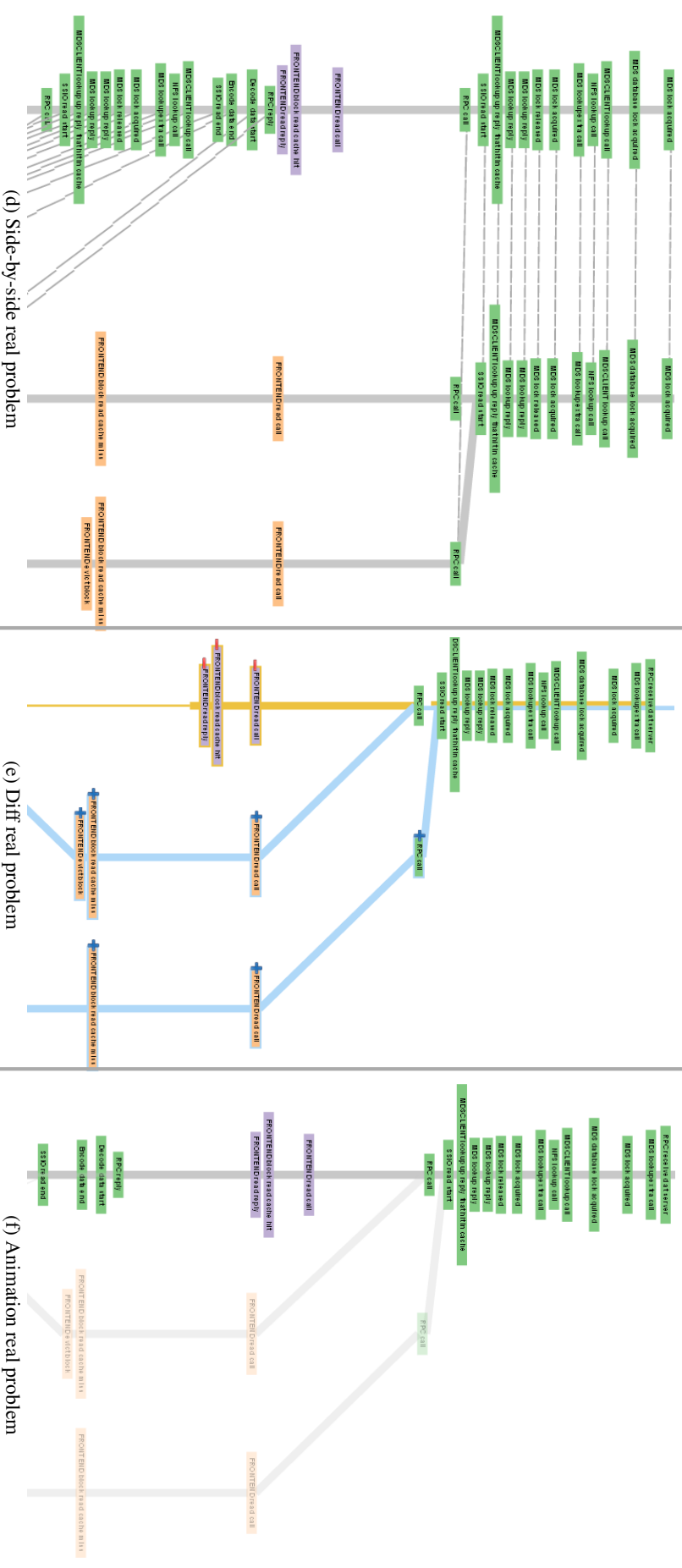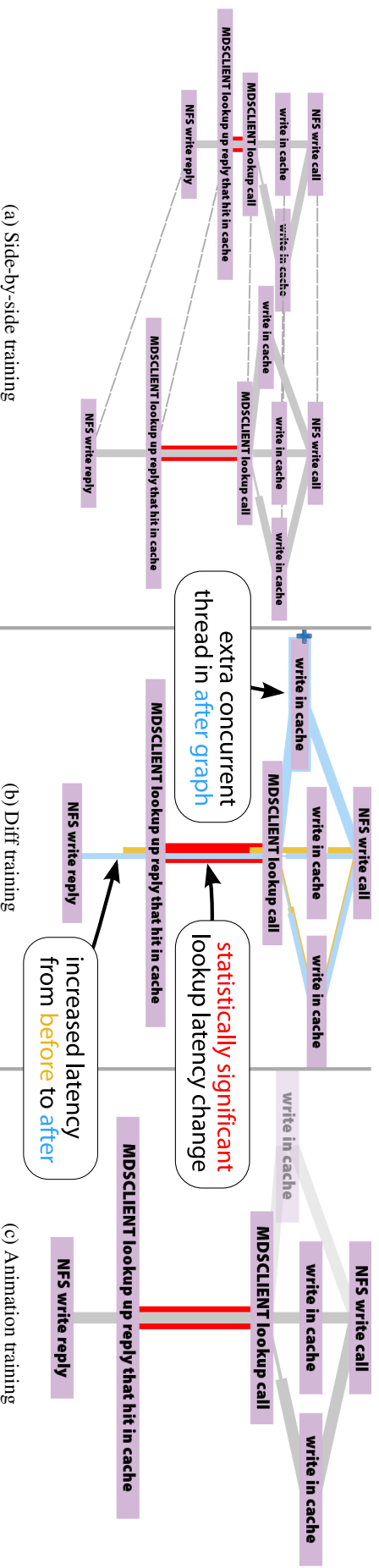
Fig. 4. **Three interfaces.** This diagram illustrates the three approaches to visualizing differences in request-flow graphs that we compare in this study. Figures a, b, and c show the interfaces applied to a mocked-up problem that was used to train participants (slightly modified for clarity on paper). Figures d, e, and f show the interfaces applied to a portion of one of the real-world problems that was presented to participants.

(a) Side-by-side training

(b) Diff training

(c) Animation training

(d) Side-by-side real problem

(e) Diff real problem

(f) Animation real problem

extra concurrent thread in after graph

statistically significant lookup latency change

increased latency from before to after

To navigate the interface, users can pan the graph view by clicking and dragging or by using a vertical scroll bar. In large graphs, this allows for movement in the neighborhood of the current view or rapid traversal across the entire graph. By using the wheel on a mouse, users can zoom in and out, up to a limit. We employ rubber-banding for both the traversal and zoom features to prevent the interface from moving off the screen or becoming smaller than the viewing window.

For all of the interfaces, edge lengths are drawn using a sigmoid-based scaling function that allows both large and small edge latencies to be visible on the same graph. Statistically significant edge latency changes are highlighted with a bold red outline. When graphs contain join points, or locations where multiple parallel paths converge at the same node, one path may have to wait for another to complete. Our interfaces illustrate the distinction between *actual* latency and waiting time by using thinner lines for the latter (see the "write in cache" to "MDSCLIENT lookup call" edge in Figures 4a-c for an example).

Each interface also has an annotation mechanism that allows users to add marks and comments to a graph comparison. These annotations are saved as an additional layer on the interface and can be restored for later examination.

4.3 Interface Example

To better illustrate how these interfaces show differences, the example of diff shown in Figure 4b is annotated with the three key differences it is meant to reveal. First, the after graph contains an extra thread of concurrent activity (outlined in blue and marked with a plus sign). Second, there is a statistically significant change in metadata lookup latency (highlighted in red). Third, there is a large latency change between the lookup of metadata and the request's reply. These observations localize the problem to those system components involved in the changes and thus provide starting points for developers' diagnosis efforts.

5 USER STUDY OVERVIEW & METHODOLOGY

We compared the three approaches via a between-subjects user study, in which we asked participants to complete five assignments using our interfaces. Each assignment asked participants to find key performance-affecting differences for a before/after request-flow graph pair obtained from Ursa Minor [1]. Four of the five assignments used graphs output by Spectroscope for real problems that were either observed in the system or injected into it. These problems are described further in Sambasivan et al. [32].

5.1 Participants

Our interfaces' target users are the developers of the distributed system being diagnosed. As our example tasks come from Ursa Minor, we recruited the seven Ursa Minor developers to whom we had access as expert participants. In addition, we recruited six professional distributed-system developers from Google. We refer to the Ursa Minor and Google developers collectively as *professionals*.

Many of our professional participants are intimately familiar with more traditional diagnosis techniques, potentially biasing their responses to our user-study questions. For a wider perspective, we recruited additional participants by advertising in undergraduate and graduate classes on distributed systems and posting fliers on and around our campus. Potential participants were required to demonstrate (via a pre-screening questionnaire) knowledge of key undergraduate-level distributed systems concepts. Of the 33 volunteers who completed the questionnaire, 29 were deemed eligible; the first 13 to respond were selected. Because all of the selected participants were graduate students in computer science, electrical and computer engineering, or information networking, we refer to them as *student* participants.

During the user study, each participant was assigned, round-robin, to evaluate one of the three approaches. Table 1 lists the participants, their demographic information, and the interface they were assigned. We paid each participant $20 for the approximately 1.5-hour study.

5.2 Creating before/after graphs

Each assignment required participants to identify salient differences in a before/after graph pair. To limit the length of the study, we modified

Table 1. **Participant demographics.** Our 26 participants included 13 professional distributed systems developers and 13 graduate students familiar with distributed systems. The ID encodes the participant group (P=professional, S=student) and the assigned interface (S=side-by-side, D=diff, A=animation). Average ages were 35 (professionals) and 25 (students).

ID	Gender	Age	Interface	ID	Gender	Age	Interface
PS01	M	26	S	SS01	F	23	S
PS02	M	33	S	SS02	M	21	S
PS03	M	38	S	SS03	M	28	S
PS04	M	44	S	SS04	M	29	S
PS05	M	30	S	SD05	M	35	D
PD06	M	37	D	SD06	M	22	D
PD07	M	44	D	SD07	M	23	D
PD08	M	37	D	SD08	M	23	D
PD09	M	28	D	SD09	M	25	D
PA10	M	33	A	SA10	F	26	A
PA11	M	26	A	SA11	M	23	A
PA12	M	40	A	SA12	M	22	A
PA13	M	34	A	SA13	M	23	A
(a) Professionals				(b) Students			

the real-problem graph pairs slightly to remove a few differences that were repeated many times. The only synthetic before/after pair was modified from a real request-flow graph observed in the system. Table 2 describes the various assignments and their properties.

To make the before/after graphs easier for participants to understand, we changed node labels, which describe events observed during request processing, to more human-readable versions. For example, we changed the label "e10__t3__NFS_CACHE_READ_HIT" to "Read that hit in the NFS server's cache." The original labels were written by Ursa Minor developers and only have meaning to them. Finally, we omitted numbers indicating edge lengths from the graphs to ensure participants used visual properties of our interfaces to find differences.

5.3 User study procedure

The 1.5 hour study consisted of a 10-15 minute training period, followed by three evaluation phases: guided questions, emulation of real diagnoses, and interface comparison. Participants were encouraged to think aloud throughout the study.

5.3.1 Training

In the training phase, participants were shown an Ursa Minor diagram (similar to the one in Figure 2). They were only required to understand that the system consists of four components that can communicate over the network. We also provided a sample request-flow graph and

Table 2. **Before/after graph-pair assignments.** Assignments 1–3 were used in the guided questions phase (G); 4 and 5 were used to emulate real diagnoses (E). Four of the five assignments used graphs from the output of Spectroscope for real problems seen in Ursa Minor.

Phase	Assignment and type	Differences	Before/after graph sizes (nodes)
G	1/Real	4 statistically sig. 5 other edge latency	122/122
	2/Real	1 structural	3/16
	3/Synth.	4 statistically sig. 2 other edge latency 3 structural	94/128
E	4/Real	4 structural	52/77
	5/Real	2 structural	82/226

described the meaning of nodes and edges. Finally, we trained each participant on her assigned interface by showing her a sample before/after graph (identical to those shown in Figures 4(a-c)) and guiding her through tasks she would have to complete in latter parts of the study. Participants were given ample time to ask questions and told we would be unable to answer further questions after the training portion.

5.3.2 Finding differences via guided questions

In this phase of the study, we guided participants through the process of identifying differences, asking them to complete five focused tasks for each of three assignments. Rows 1–3 of Table 2 describe the graphs used for this part of the study.

TASK 1: *Find any edges with statistically significant latency changes*. This task required participants to find all of the graph edges highlighted in red (i.e., those automatically identified by Spectroscope as having statistically significant changes in latency distribution).

TASK 2: *Find any other edges with latency changes worth investigating*. Spectroscope will not identify all edges worth investigating. For example, edges with large changes in average latency that also exhibit high variance will not be identified. This task required participants to find edges with notable latency changes not highlighted in red.

TASK 3: *Find any groups of structural changes*. Participants were asked to identify added or deleted nodes. To reduce effort, we asked them to identify these changes in contiguous groups, rather than noting each changed node individually.

TASK 4: *Describe in a sentence or two what the changes you identified in the previous tasks represent*. This task examines whether the interface enables participants to quickly develop an intuition about the problem in question. For example, many of the edge latency changes presented in assignment 1 indicate a slowdown in network communication between machines. Identifying these themes is a crucial step toward understanding the root cause of the problem.

TASK 5: *Of the changes you identified in the previous tasks, identify which one most impacts request response time*. The difference that most affects response time is likely the one that should be investigated first when attempting to find the root cause. This task evaluates whether the interface allows participants to quickly identify this key change.

5.3.3 Emulating real diagnoses

In the next phase, participants completed two additional assignments. These assignments were intended to emulate how the interfaces might be used in the wild, when diagnosing a new problem for the first time. For each assignment, the participant was asked to complete tasks 4 and 5 only (as described above). We selected these two tasks because they most closely align with the questions a developer would ask when diagnosing an unknown problem.

After this part of the study, participants were asked to agree or disagree with two statements using a five-point Likert scale: "I am confident my answers are correct" and "This interface was useful for solving these problems." We also asked them to comment on features of the interface they liked or disliked and to suggest improvements.

5.3.4 Interface comparison

Finally, to get a more direct sense of what aspects of each approach were useful, we showed participants an alternate interface. To avoid fatiguing participants and training effects, we did not ask them to complete the assignments and tasks again; instead we asked them to briefly consider (using assignments 1 and 3 as examples) whether the tasks would be easier or harder to complete with the second interface, and to explain which features of both approaches they liked or disliked. Because our pilot study suggested animation was most difficult to use, we focused this part of the study on comparing side-by-side and diff.

5.4 Scoring criteria

We evaluated participants' responses by comparing them to an "answer key" created by an Ursa Minor developer who had previously used Spectroscope to diagnose the real problems used in this study. Tasks 1–3, which asked for multiple answers, were scored using precision/recall. *Precision* measures the fraction of a participant's answers that were also in the key. *Recall* measures the fraction of all answers in the key identified by the participant. Note that it is possible to have high precision and low recall—for example, by identifying only one change out of ten possible ones. For task 3, participants who marked any part of a correct group were given credit.

Tasks 4 and 5 were graded as correct or incorrect. For both, we accepted multiple possible answers. For example, for task 4 ("identify what changes represent"), we accepted an answer as correct if it was close to one of several possibilities, corresponding to different levels of background knowledge. In one assignment, several students identified the changes as representing extra cache misses in the after graph, which we accepted. Some participants with more experience explicitly identified that the after graph showed a read-modify write, a well-known bane of distributed storage system performance.

We also captured completion times for the five quantitative tasks. For completion times as well as precision/recall, we used the Kruskal-Wallis test to establish differences among all three interfaces, then pairwise Wilcoxon Rank Sum tests (chosen a priori) to separately compare the animation interface to each of side-by-side and diff. We recorded and analyzed participants' comments from each phase as a means to better understand the strengths and weaknesses of each approach.

5.5 Limitations

Our methodology has several limitations. Most importantly, it is difficult to fully evaluate visualization approaches for helping developers diagnose problems without going through the entire process of debugging a real, complex problem. However, such problems are often unwieldy and can take days to diagnose. As a compromise, we designed tasks to test whether our interfaces enable participants to understand the gist of the problem and identify starting points for diagnosis.

Deficiencies in our interface implementations may skew participants' notions of which approaches work best for various scenarios. We explicitly identify such cases in our evaluation and suggest ways for improving our interfaces so as to avoid them in the future.

We stopped recruiting participants when their qualitative comments converged, leading us to believe we had enough information to identify the useful aspects of each interface. However, our small sample size may limit the generalizability of our quantitative results.

Many of our participants were not familiar with statistical significance and, as such, were confused by the wording of some of our tasks (especially tasks 1 and 2). We discuss this in more detail in Section 7.

Our participants skewed young and male. To some extent this reflects the population of distributed-systems developers and students, but it may limit the generalizability of our results somewhat.

6 USER STUDY RESULTS

No single approach was best for all users and types of graph differences. For example, side-by-side was preferred by novices, and diff was preferred by advanced users and experts. Similarly, where side-by-side and diff proved most useful for most types of graph differences, animation proved better than side-by-side and diff for two very common types. When one of our participants (PD06) was asked to pick his preferred interface, he said, "If I had to choose between one and the other without being able to flip, I would be sad." When asked to contrast side-by-side with diff, SS01 said, "This is more clear, but also more confusing." Section 6.1 compares the approaches based on participants' quantitative performance on the user study tasks. Sections 6.2 to 6.4 describe our observations and participants' comments about the various interfaces and, based on this data, distill the approaches best suited for specific graph difference types and usage modes.

6.1 Quantitative results

Figure 5 shows completion times for each of the three interfaces. Results for individual tasks, aggregated over all assignments, are shown (note that assignments, as shown in Table 2, may contain one or multiple types of differences). Participants who used animation took longer to complete all tasks compared to those who used side-by-side or diff, corroborating the results of several previous studies [4, 13, 31]. Median completion times for side-by-side and diff are similar for most tasks.

The observed differences between animation and the other interfaces are statistically significant for task 1 ("identify statistically significant changes") and task 4 ("what changes represent").[1] The observed trends are similar when students and professionals are considered separately, except that the differences between animation and the other interfaces are less pronounced for the latter.

Figure 6a shows the precision, recall, and accuracy results for each of the three interfaces. Our results are not statistically significant, but do contain artifacts worth describing. Overall, both side-by-side and diff fared well, and their median scores for most tasks are similar for precision, recall, and accuracy. Notable exceptions include recall for task 2 ("find other latency changes") and recall for task 3 ("identify structural changes"), for which diff performed better. Overall, both diff and animation exhibit much higher variation in scores than side-by-side. Though animation's median scores are better than or comparable to the other interfaces for tasks 3, 4, and 5, its scores are worse for precision for task 1 and recall for task 2.

Figures 6b and 6c show the results broken down by participant type. No single interface yielded consistently higher median scores for either group. Though professionals performed equally well with diff and side-by-side for many tasks, their scores with diff are higher for tasks 2 and 3 and higher with side-by-side for task 4. Students' median scores were higher with side-by-side for task 2 and task 5 and higher for recall with diff for task 1 and task 3. Also, students' diff scores exhibit significantly more variation than side-by-side, perhaps because not all of them were familiar with text-based diff tools, which are often used by professionals for source code-revision control. For professionals, animation's median scores are almost never higher than side-by-side. Students had an easier time with animation. For them, animation's median scores are higher than diff and side-by-side for task 2 (precision), task 4, and task 5. Animation's median score is higher than side-by-side for task 3 (recall).

Figure 7 shows Likert-scale responses to the questions "I am confident my answers are correct" and "This interface was useful for answering these questions." Diff and side-by-side were tied in the number of participants that strongly agreed or agreed that they were confident in their answers (5 of 9, or 56%). However, where one side-by-side user strongly agreed, no diff users did so. Only one animation user (of eight; 12.5%) was confident in his answers, so it is curious that animation was selected as the most useful interface. We postulate this is because participants found animation more engaging and interactive than the other interfaces, an effect also noted by other studies [13, 31].

[1] Task 1: χ^2=6.7, df=2, p=0.04 (omnibus), W=14, p=0.04 (side-anim), W=12, p=0.02 (diff-anim). Task 4: χ^2=6.05, df=2, p<0.05 (omnibus), W=14, p=0.04 (side-anim), W=14, p=0.04 (diff-anim).

Fig. 6. **Precision/recall scores.** The boxplots show precision, recall, and accuracy scores for individual tasks, aggregated across all assignments. Boxes extend from the first quartile to the third quartile, and whiskers to the range of the data. Colored triangles represent single outliers.

6.2 Side-by-side

Participants liked the side-by-side interface because it was the most straightforward one. It showed the true response times (i.e., overall latencies) of both graphs, enabling participants to quickly get a sense of how much performance had changed. Correspondence lines clearly showed matching nodes in each graph. Also, this interface allowed independent analyses of both graphs, which our professional participants said was important. Comparing diff to side-by-side, PD08 said "it's [side-by-side] a lot easier to tell what the overall latency is for each operation. . . . [the nodes are] all put together without any gaps in the middle." SD09 said, "With [side-by-side], I can more easily see this is happening here before and after. Without the dashed lines, you can't see which event in the previous trace corresponds to the after trace." These sentiments were echoed by many others (e.g., SD06, SD07, PD07).

Our side-by-side interface suffers from two key drawbacks. First, it is difficult to identify differences when before/after graphs differ significantly because corresponding nodes become farther apart. PS01 complained that "the points that should be lining up are getting farther and farther away, so it's getting more difficult to compare the two." PD06 complained that it was more difficult to match up large changes since the matching counterpart could be off the screen. Similar complaints were voiced by other participants (e.g., PS02, SS02, PS04). Adding the ability to pin one graph relative to another to our side-by-
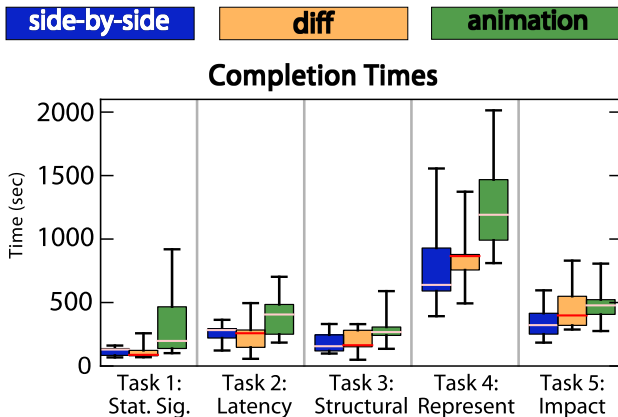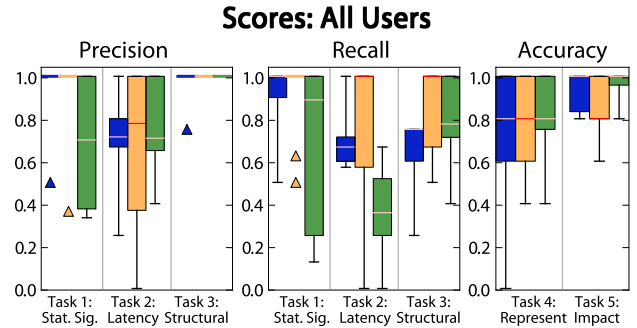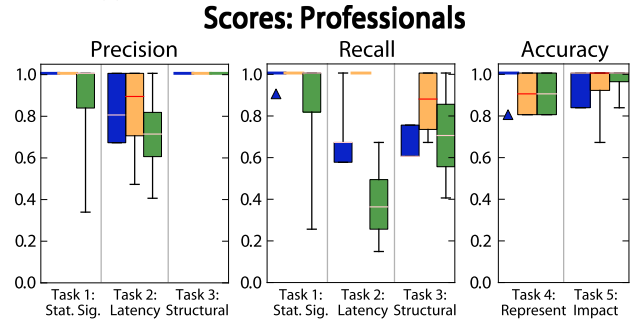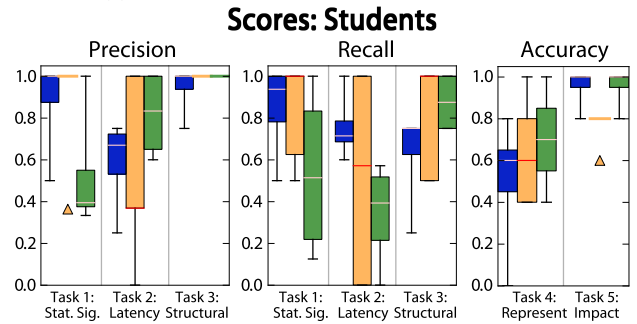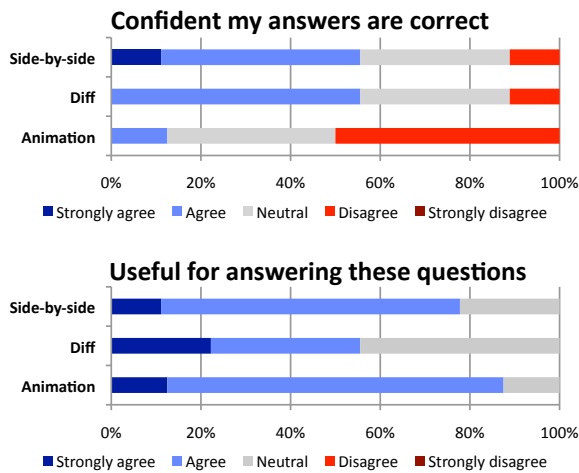

Fig. 5. **Completion times for all participants.** The boxplots show completion times for individual tasks, aggregated across all assignments. Boxes extend from the first quartile to the third quartile, and whiskers to the range of the data.

Confident my answers are correct

Useful for answering these questions

Fig. 7. **Likert responses, by condition.** Each participant was asked to respond to the statements "I am confident my answers are correct" and "The interface was useful for answering these questions."

side implementation would limit vertical distance between differences. However, horizontal distance, which increases with the number of concurrent threads in each request, would remain.

Second, when nodes are very close to another, correspondence lines became too cluttered and difficult to use. This led to complaints from several participants (e.g., PS03, SS01, SS03, PA13). To cope, SS03 and PS05 gave up trying to identify matching nodes between the graphs and instead identified structural differences by determining if the number of correspondence lines on the screen matched the number of nodes visible in both the before and after graph. Modifying our side-by-side interface to draw correspondence lines only at the start of a contiguous run of matching nodes would help reduce clutter, but would complicate edge latency comparisons.

Based on participants' comments above and our observations, Table 3 shows the use cases for which we believe side-by-side is the best of the three approaches. As shown in Table 3, side-by-side's simple approach works best for aiding comprehension. However, due to potential for horizontal skew and clutter, it is not the best approach for identifying any type of difference.

6.3 Diff

Participants' comments about our diff interface were polarized. Professionals and more advanced students preferred diff's compactness, whereas others were less decisive. For example, PS03 claimed diff's compact representation made it easier for him to draw deductions. Indeed, unlike side-by-side, diff always shows differences right next to each other, making it easier to find differences when before and after graphs have diverged significantly. Also, by placing differences right next to each other, diff allows for easier identification of smaller structural and edge latency changes. In contrast, SS04 said, "[Side-by-

side] may be more helpful than [diff], because this is not so obvious, especially for structural changes."

Though participants rarely made explicit comments about finding diff challenging to use, we found that diff encouraged incorrect mental models in student participants. For example, SD08 and SD09 confused structural differences within a single thread of activity with a change in the amount of concurrency. It is easy to see why participants might confuse the two, as both are represented by fan-outs in the graph.

We postulate that participants' comments about diff vary greatly because its compact representation requires more knowledge about software development and distributed systems than that required by the more straightforward side-by-side interface. Additionally, many of our professionals are familiar with diff tools for text, which would help them understand our graph-based diff technique more easily.

Since it places differences close together, Table 3 lists diff as the best approach for showing edge latency changes. However, because it encourages poor mental models for structural differences, it is not the best approach for showing concurrency changes and intra-thread event changes.

6.4 Animation

Our participants often struggled when using our animation interface. With this interface, all differences between the two graphs appear and disappear at the same time. This combined movement causes two problems. First, it creates a large amount of visual flux when both structural and edge latency differences are present within the same thread of activity. This confused users and prevented them from accurately identifying differences. PA11 complained, "Portions of graphs where calls are not being made in the after trace are fading away while other nodes move on top of it and then above it ... it is confusing." These sentiments were echoed by many other participants (e.g., SA11, PA12, SA10, PA13).

Second, the large amount of combined movement that results when multiple difference types are present in the same thread prevented participants from establishing static reference points for gauging the impact of a given difference. SA10 told us: "I want to... pick one node and switch it between before and after. [But the same node] in before/after is in a different location completely." SA12 said he did not like our animation interface because of the lack of consistent reference points. "If I want to measure the size of an edge, if it was in the same position as before... then it'd be easy to see change in position or length."

Staged animation techniques [20], in which differences are animated in one at a time, could reduce combined movement, lower visual flux, and allow users to establish reference points. However, significant research is needed to understand how to effectively stage animations for graphs that exhibit both structural and edge latency changes. Many graph animation visualizations do not use staging and only recent work has begun to explore where such basic approaches fail [16].

Another negative aspect of animation (staged or not) is that it suggests false intermediate states. As a result, SA13 interpreted our interface's animation sequence as a timeline of changes and listed this as a feature he really liked. PA13 told us we should present a toggle instead of a slider so as to clarify that there are only two states.

Table 3. **Most useful approaches for aiding overall comprehension and helping identify the various types of graph differences contained in the user study assignments.** These results are based on our qualitative observations and participants' comments. A ✓ indicates the best approach for a particular category, whereas a ✗ indicates an approach poorly suited to a particular category. Side-by-side is best for aiding overall comprehension because of its straightforward presentation. Diff is best for showing edge latency changes because it places such changes right next to one another. Animation is best for showing structural differences (i.e., intra-thread event changes and concurrency changes) due to the blinking effect it creates for them. Due to their various drawbacks, no single approach is best for showing multiples types of differences within a single thread of activity.

	Comprehension		Difference identification			
	Shows overall latency change	Supports independent analyses	Concurrency change	Intra-thread event change	Edge latency change	Intra-thread mixed
Side	✓	✓				
Diff		✗			✓	
Anim		✓	✓	✓		✗

Despite the above drawbacks, animation excels at showing graphs with only structural differences, such as changes to events within threads (i.e., intra-thread event changes) and concurrency changes. Participants can easily find reference points for such difference by looking for the beginning and end of the structural change (or a contiguous group of them). Additionally, structural differences by themselves, especially concurrency-related ones, do not create a large amount of visual flux because they tend to affect distinct, segmented portions of the graphs. As such, instead of a cacophony of movement, the effect when structural differences are animated is a pleasing blinking effect in which distinct portions of the graph appear and disappear, allowing users to identify such differences easily. For one such assignment, both PA11 and PA12 told us the structural difference was very clear with animation. Other studies have also noted that animation's effectiveness increases with increasing separation of changed items or decreasing visual flux [4, 16].

Due to the blinking effect it creates, Table 3 lists animation as the best approach for showing structural differences. However, the problems caused by combined movement make it the worst interface for showing both edge latency and structural differences within a single thread of activity (i.e., intra-thread mixed changes).

7 DESIGN LESSONS & RESEARCH CHALLENGES

In addition to yielding insights about which visualization approaches are best suited to different scenarios, our user study experiences have helped us identify several important design guidelines for future diagnosis visualizations. This section describes the most important ones, and highlights associated research challenges.

Provide anchor points to help reduce skew and combined movement: Many users struggled with the increasing skew in the side-by-side layout, as well as the inability to quickly trace a correspondence from one graph to another (e.g., PS02, SA10, and PS05). Users also struggled with the animation interface when both edge latency and structural differences were present within the same thread because it animated all of them together. A future interface might anchor the comparison in multiple or user-selectable locations to mitigate this problem. However, there are subtleties involved in choosing and using anchor points.

One option, as requested by most of our participants (e.g., PA12 and PD08), is to anchor the comparison at a user-selectable location. Another is to re-center the graph as users scroll through it. However, both techniques distort the notion that time flows downward, and neither would reduce horizontal distance or clutter. Approaches that restructure the comparison to minimize the horizontal distance between matched nodes are an interesting research opportunity.

For the animation technique, anchoring in multiple locations could be achieved by staging changes. Questions of ordering immediately arise: structural changes might be presented before, after, or between latency changes. The choice is non-obvious. For example, it is not clear whether to animate structural and latency changes together when the structural change *causes* the latency change or even how to algorithmically determine such cases (see Figure 1 for an example).

Selectively reduce diagnosis output complexity: Even when they were navigable, graphs with hundreds of nodes posed an obstacle to understanding the output at a high level. While the straightforward approach to zooming we implemented for this study allows the entire graph to be seen, it does not provide intuition into the graph's meaning when zoomed out. Therefore, users needed to scroll through graphs while zoomed in, a complaint for multiple users (e.g., SD05, SA10).

Further work is needed to investigate options for a higher-level view of the graph that could be shown instead. Some options for semantic zooming based on such a view include coalescing portions of the comparison that are the same in both graphs, or grouping sequences of similar operations (mentioned by several participants, including SD09, PS02, and PD09). Determining which portions to coalesce and defining "similar" operations are non-trivial problems, and solutions may require user guidance or automated techniques to learn meaningful groupings.

Include strong annotation mechanisms: Annotation was used in this study to record answers, but has the potential to be a valuable tool for collaborative debugging. Developing and debugging a large system involves multiple components built and maintained by different parties, many without knowledge of the internal workings of components that might contain a problem. Users could make annotations for other developers or for documentation. In fact, several professional participants from Google listed our annotation mechanism as a strength of the interfaces (e.g., PA13, PS04, and PD08). PS04 said "[I] really like the way you added the annotation....So other people who are later looking at it can get the benefit of your analysis." Supporting cooperative diagnosis work with an annotation interface, such as that used in Rietveld [29] for code reviews, is an interesting avenue of future work.

Match automation to users' expectations: Like several other diagnosis tools, Spectroscope uses statistical significance as the bar for automatically identifying differences, because it bounds wasted developer effort by limiting the expected number of false positives. However, many of our participants did not have a strong background in statistics and so mistook "statistically significant" to mean "large changes in latency." They did not know that variance affects whether an edge latency change is deemed statistically significant. This generated confusion and accounted for lower than expected scores for some tasks. For example, some participants (usually students) failed to differentiate between task 1 and task 2, and a few refused to mark a change as having the most impact unless it was highlighted in red (as statistically significant). Trying to account for why one particularly large latency change was not highlighted, SA10 said, "I don't know what you mean by statistically significant. Maybe it's significant to me." These concerns were echoed by almost all of our participants, and demonstrate that automation must match users' mental models. Statistics and machine learning techniques can provide powerful automation tools, but to take full advantage of this power—which becomes increasingly important as distributed systems become more complex—developers must have the right expectations about how they work. Both better techniques and more advanced training may be needed to achieve this goal.

8 SUMMARY & NEXT STEPS

For tools that automate aspects of problem diagnosis to be useful, they must present their results in a manner developers find clear and intuitive. This paper compares the relative merits of three commonly used visualization approaches in the context of presenting the results of one promising automated problem localization technique. Via a 26-participant user study, we found that each approach has unique strengths for different distributed systems problem types, developer usage modes, and users.

Since each of the approaches provided unique benefits, we believe the next step of our research will involve developing an integrated interface that allows users to dynamically switch between side-by-side, diff, and animation. We suspect that empowering users to view the same output from multiple perspectives will enable insights not possible with any single approach. As part of developing this interface, we believe it is important to explore how to stage animation to reduce combined movement and also to investigate mechanisms for semantic zooming of large, complex graphs.

Finally, we are also interested exploring whether our insights about the various approaches could help guide visualizations in other domains that use directed acyclic graphs (or similar types of graphs). Examples include comparing software versions [10], business-process graphs [3] and general workflows [33].

REFERENCES

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. Strunk, E. Thereska, M. Wachs, and J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proc. FAST*, 2005. Cited on pages 2 and 5.

[2] B. Alper, B. Bach, N. H. Riche, T. Isenberg, and J.-D. Fekete. Weighted graph comparison techniques for brain connectivity analysis. In *Proc. CHI*, 2013. Cited on page 3.

[3] K. Andrews, M. Wohlfahrt, and G. Wurzinger. Visual graph comparison. In *Proc. IV*, 2009. Cited on page 9.

[4] D. Archambault, H. C. Purchase, and B. Pinaud. Animation, small multiples, and the effect of mental map preservation in dynamic graphs. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):539–552, Apr. 2011. Cited on pages 2, 6, and 9.

[5] D. Archambault, H. C. Purchase, and B. Pinaud. Difference map readability for dynamic graphs. In *Proc. GD*, 2011. Cited on pages 2 and 3.

[6] F. Beck and S. Diehl. Visual comparison of software architectures. In *Proc. SOFTVIS*, 2010. Cited on page 3.

[7] N. G. Belmonte. The Javascript Infovis Toolkit. http://www.thejit.org. Cited on page 3.

[8] H. Bunke. Graph matching: theoretical foundations, algorithms, and applications. In *Proc. VI*, 2000. Cited on page 3.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. OSDI*, 2006. Cited on page 2.

[10] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proc. SOFTVIS*, 2003. Cited on page 9.

[11] R. Dijkman, M. Dumas, and L. García-Bañuelos. Graph matching algorithms for business process model similarity search. In *Proc. BPM*, 2009. Cited on page 3.

[12] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz and Dynagraph – static and dynamic graph drawing tools. In *Graph Drawing Software*, Mathematics and Visualization, pages 127–148. Springer-Verlag, 2004. Cited on page 3.

[13] M. Farrugia and A. Quigley. Effective temporal graph layout: a comparative study of animation versus static display methods. *Information Visualization*, 10(1):47–64, Jan. 2011. Cited on pages 2, 3, 6, and 7.

[14] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129, Jan. 2010. Cited on page 3.

[15] Gapminder. http://www.gapminder.org. Cited on page 3.

[16] S. Ghani, N. Elmqvist, and J. S. Yi. Perception of animated node-link diagrams for dynamic graphs. *Computer Graphics Forum*, 31(3):1205–1214, June 2012. Cited on pages 8 and 9.

[17] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4):289–309, Oct. 2011. Cited on page 2.

[18] J. A. G. Gomez, A. Buck-Coleman, C. Plaisant, and B. Shneiderman. Interactive visualizations for comparing two trees with structure and node value changes. Technical Report HCIL-2012-04, University of Maryland, 2012. Cited on page 3.

[19] M. Hascoët and P. Dragicevic. Interactive graph matching and visual comparison of graphs and clustered graphs. In *Proc. AVI*, 2012. Cited on page 3.

[20] J. Heer and G. Robertson. Animated transitions in statistical data graphics. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1240–1247, Nov. 2007. Cited on page 8.

[21] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proc. SIGCOMM*, 2009. Cited on pages 1, 2, and 3.

[22] Z. Liu, B. Lee, S. Kandula, and R. Mahajan. NetClinic: interactive visualization to enhance automated fault diagnosis in enterprise networks. In *Proc. VAST*, 2010. Cited on pages 1 and 3.

[23] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and merging files with GNU diff and patch*. Network Theory Ltd, 2002. Cited on page 3.

[24] F. Mansmann, F. Fischer, D. A. Keim, and S. C. North. Visual support for analyzing network traffic and intrusion detection events using TreeMap and graph representations. In *Proc. CHiMiT*, 2009. Cited on pages 1 and 3.

[25] A. G. Melville, M. Graham, and J. B. Kennedy. Combined vs. separate views in matrix-based graph analysis and comparison. In *Proc. IV*, 2011. Cited on page 3.

[26] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou. TreeJuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. *ACM Transactions on Graphics*, 22(3):453–462, 2003. Cited on page 3.

[27] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. NSDI*, 2012. Cited on pages 1 and 2.

[28] A. J. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, Feb. 2012. Cited on pages 1 and 3.

[29] Rietveld code review system. http://code.google.com/p/rietveld/. Cited on page 9.

[30] G. Robertson, K. Cameron, M. Czerwinski, and D. Robbins. Polyarchy visualization: visualizing multiple intersecting hierarchies. In *Proc. CHI*, 2002. Cited on page 3.

[31] G. Robertson, R. Fernandez, D. Fisher, B. Lee, and J. Stasko. Effectiveness of animation in trend visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1325–1332, Nov. 2008. Cited on pages 2, 3, 6, and 7.

[32] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proc. NSDI*, 2011. Cited on pages 1, 2, and 5.

[33] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with VsTrails. In *Proc. SIGMOD*, 2008. Cited on page 9.

[34] H. Sharara, A. Sopan, G. Namata, L. Getoor, and L. Singh. G-PARE: a visual analytic tool for comparative analysis of uncertain graphs. In *Proc. VAST*, 2011. Cited on page 3.

[35] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, Apr. 2010. Cited on page 2.

[36] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, Feb. 1981. Cited on page 3.

[37] Twitter Zipkin. https://github.com/twitter/zipkin. Cited on page 2.