

Visualizing request-flow comparison to aid performance diagnosis in distributed systems

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

CMU-PDL-12-102

May 2012

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Distributed systems are complex to develop and administer, and performance problem diagnosis is particularly challenging. When performance decreases, the problem might be in any of the system's many components or could be a result of poor interactions among them. Recent research has provided the ability to automatically identify a small set of most likely problem locations, leaving the diagnoser with the task of exploring just that set. This paper describes and evaluates three approaches for visualizing the results of a proven technique called "request-flow comparison" for identifying likely causes of performance decreases in a distributed system. Our user study provides a number of insights useful in guiding visualization tool design for distributed system diagnosis. For example, we find that both an overlay-based approach (e.g., diff) and a side-by-side approach are effective, with tradeoffs for different users (e.g., expert vs. not) and different problem types. We also find that an animation-based approach is confusing and difficult to use.

Acknowledgements: We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, NEC Laboratories, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, and Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by a Google research award, NSF grant #CNS-1117567, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). Ilari Shafer is supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense.

Keywords: distributed systems, performance diagnosis, request-flow comparison, user study, visualization

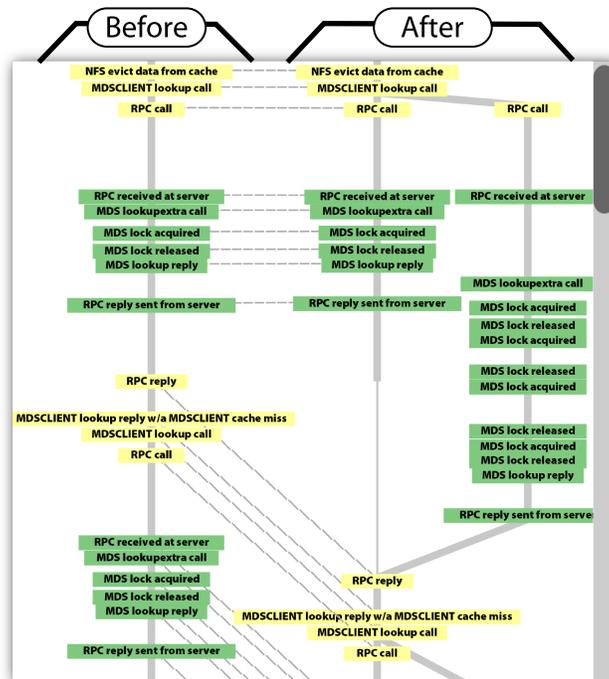


Figure 1: Comparing request-flow graphs. This side-by-side visualization, one of three interfaces we evaluate, illustrates the output of a diagnosis technique that compares graphs (called request-flow comparison). It shows these two graphs juxtaposed horizontally, with dashed lines between matching nodes 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. See Section 2 for further detail on why this change is important and what nodes in the graph represent.

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), entertain ourselves (e.g., YouTube), and so forth. Though such distributed systems often have simple interfaces and usually respond quickly, there is great complexity 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 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 tools for automatically localizing the many possible sources of a new problem to just a few potential culprits [5, 8, 9, 19–21, 25, 30, 31, 34, 36]. These tools 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 that this human-in-the-loop model will be dominant for the foreseeable future. As such, many researchers recognize the need for localization tools to present their results as clearly as possible [26, 29]. But

apart from from a few select instances [23,26], little research has been conducted on how to do so.

As a step toward addressing this need, this paper presents a 20-person user study evaluating three interfaces we built for visualizing the results of one powerful, proven technique called “request-flow comparison” [34]. Our user study uses real problems from a real distributed system. Request-flow comparison compares how the 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 request serviced 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 front-end 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 distributed storage system. Each such request flow can be represented as a graph, and comparing the before and after graphs can provide significant insight into performance degradations.

Although request-flow comparison has been used to diagnose real problems observed in the Ursa Minor distributed storage system [1] as well as certain Google services, its utility has been limited by a clunky interface that presents results using text files and unsophisticated DOT [11] graphs that must be manually and painstakingly compared with each other. The goal of this study is to identify what visualization techniques work best for presenting the results of request-flow comparison to their intended audience—developers and people knowledgeable about distributed systems.

The interfaces we compared all try to show relevant differences between before-and-after pairs of directed acyclic graphs, which are the output of request-flow comparison. We built our own interfaces because of domain-specific requirements that precluded off-the-shelf solutions. For example, correspondences between nodes of before-after pairs are not known a priori, so a significant challenge involved creating heuristics to identify them. The *side-by-side* interface shows both graphs, adding correspondence lines between nodes that represent the same activity in both graphs (see Figure 1). *Diff* presents a compact view by overlaying the graphs and highlighting important differences. Finally, *animation* attempts to clarify differences by rapidly switching between the two graphs.

Our user study results show *diff* and *side-by-side* perform comparably, with *animation* faring the worst. The choice between *diff* and *side-by-side* varies depending on users’ familiarity with software development practices and with characteristics of the problem being diagnosed. Non-experts preferred *side-by-side* due to

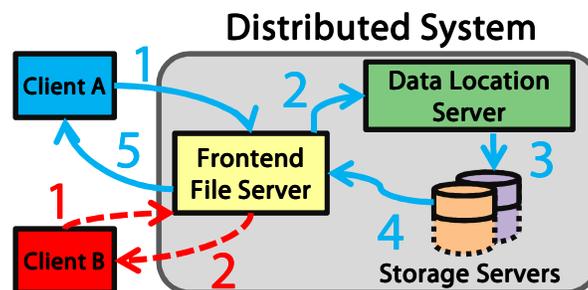


Figure 2: Distributed storage system. To read a file, clients connect to this distributed system. A frontend file server handles their requests, but may need to access other components like the data location server and storage servers. For example, Client A makes a request that requires the blue messages 1–5, while the request initiated by Client B only produces the red messages 1–2. There may be many other paths through the system. Ursa Minor, the distributed storage system discussed in this paper, has a similar architecture.

its straightforward presentation. Diff’s more compact representation was favored by experts and advanced users, but it engendered confusion in those less familiar with distributed systems. We also found that request-flow comparison’s failure modes sometimes did not match users’ expectations, highlighting the importance of choosing algorithms that match users’ mental models when creating automated diagnosis tools.

The rest of this paper is organized as follows. Section 2 provides relevant background about request-flow comparison. Section 3 describes the three interfaces and how node correspondences are determined. Section 4 describes the user study, and Section 5 describes the results. Section 6 describes key design lessons learned from the user study for further interface improvements. Section 7 describes related work and Section 8 concludes.

2 Request-flow comparison

Request-flow comparison [34] is a technique for automatically localizing the root causes of performance degradations in distributed systems, such as Ursa Minor (shown in Figure 2), GFS [14], and Bigtable [7]. It uses the insight that such degradations often manifest as changes or differences in the workflow of individual requests as they are serviced by the system. Exposing these differences and showing how they differ from previous behavior localizes the source of the problem and significantly guides developer effort.

Request-flow comparison works by comparing request-flow graphs observed during two periods: one of good performance and one of poor performance. Nodes of these directed acyclic graphs show important events observed on different components during request processing, and edges show latency between these events (see Figure 3 for an example). Request-flow comparison groups the flows observed during both periods (often numbered in the hundreds of thousands or millions) into clusters, then identifies those from the poor-performance period that appear to most contribute to the performance degradation. As output, it presents pairs of *before-and-after* graphs of these culprits, showing how they were processed before the performance change versus after the change.¹ Identifying differences between these pairs of graphs localizes the source of the problem and provides developers with starting points for their diagnosis efforts. To preserve context, entire request-flow graphs are presented with some, but not all, important differences highlighted.

This technique identifies two important types of differences. *Edge latency changes* are differences in the time required to execute successive events and represent slowdown in request processing. Request-flow comparison attempts to identify these changes automatically, using hypothesis tests to identify edges with latency distributions that have a statistically significant difference in the before and after periods. Similar tests are used in several automated diagnosis tools [19, 28, 34]. Since hypothesis tests will not identify all edges worth investigating, developers must still examine the graphs manually to find additional such divergences. *Structural changes* are differences in the causal ordering of system events. Developers must contrast the two graphs manually to identify them. Further details about request-flow comparison can be found in Sambasivan et al. [34].

3 Interface design

To compare the pairs of before/after graphs output by request-flow comparison, we built three interfaces designed to represent orthogonal approaches to representing differences. They are shown in Figure 4. The interfaces occupy three corners in the space of approaches to visualizing differences, as identified by a

¹In Sambasivan et al. [34], before graphs are called precursors and after graphs are called mutations.

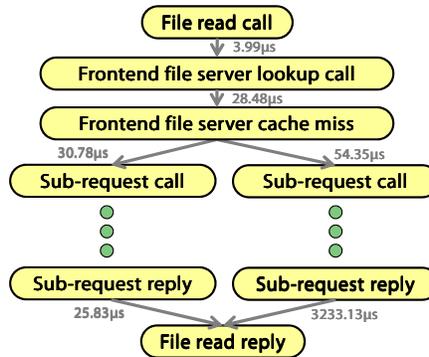


Figure 3: Example request-flow graph. This graph shows the flow of a read request through the distributed storage system shown in Figure 2. Node names represent important events observed on the various components while completing the required work. Edges show latencies between these events. Fan outs represent the start of parallel activity, and synchronization points are indicated by fan ins. Due to space constraints, only the events observed on the frontend file server are shown. The green dots abstract away messages exchanged between other components and the work done on them. Finally, the original node names, which had meaning only to the developers of the system, have been replaced with human-readable versions.

taxonomy of comparison approaches [15]. The side-by-side interface is nearly a “juxtaposition,” which presents independent layouts. The diff interface is an “explicit encoding,” which highlights the differences between the two graphs. Finally, the animation interface is closest to a “superposition” design that guides attention to changes that “blink.” All are implemented in JavaScript, and use modified libraries from the Javascript InfoVis Toolkit [6]. The rest of this section describes these interfaces.

3.1 Side-by-side

The side-by-side interface (Figures 4a and 4d) computes independent layered layouts for the before and after graphs and displays them beside each other horizontally. Nodes in the before graph are linked to corresponding nodes in the after graph by dashed lines. This interface is analogous to a parallel coordinates visualization [18], with coordinates given by the locations of the nodes in the before and after graphs. Figure 4a provides an example of this interface for two graphs containing nodes a,b,c, and d. 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 corresponding node in the other graph.

3.2 Diff

The diff interface (Figures 4b and 4e) shows a single static image in 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. Nodes that exist in both graphs are not highlighted. This structural approach is akin to the output of a contextual diff tool [24] emphasizing insertions and deletions.

We use the same orange and blue scheme to show latency changes, with edges that exist in only one graph

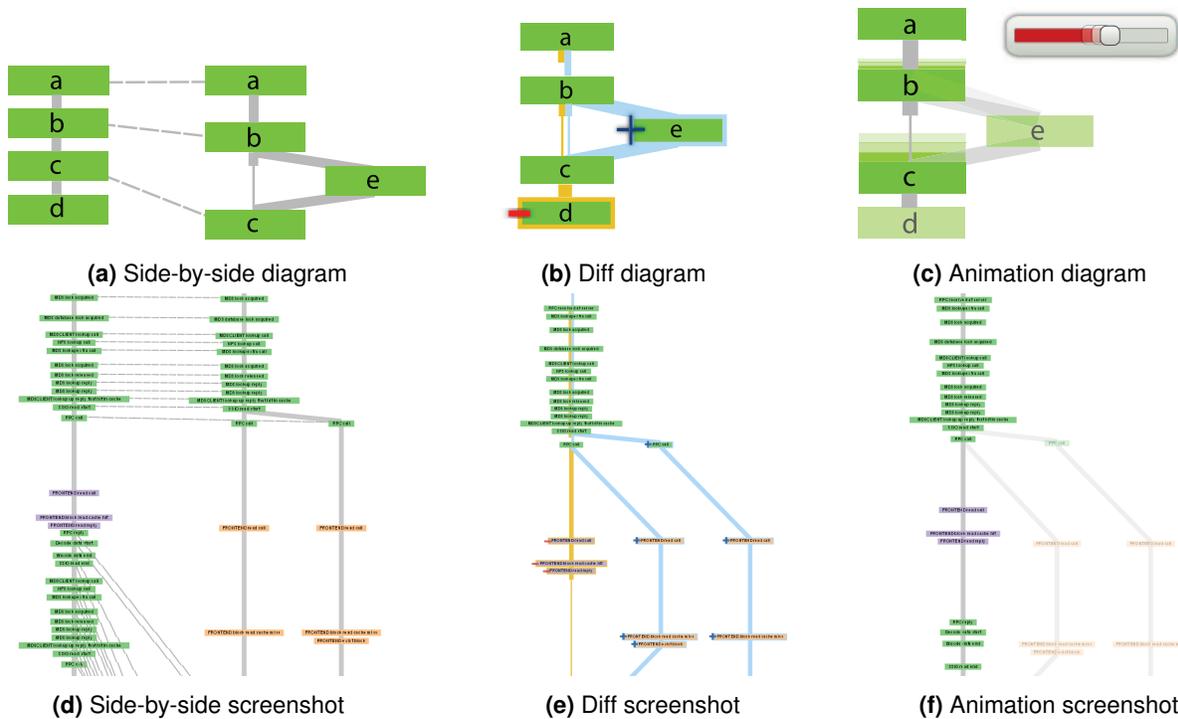


Figure 4: Three interfaces. This diagram illustrates the three approaches to visualizing differences in request-flow graphs that we compare in this study. Figures a, b, and c provide samples for small synthetic graphs. Figures d, e, and f show the interfaces applied to one of the real-world problems that was presented to users.

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 subsequent node is attached at the end of the longer line.

3.3 Animation

The animation interface (Figures 4c and 4f) provides user-controllable switching between the before and after graphs. To provide a smooth transition, we interpolate the positions of nodes between the two graphs. Nodes that exist in only one graph appear only on the appropriate terminal of the animation, becoming steadily more transparent as the animation advances and vanishing completely by the other terminal. Users can start and stop the animation, as well as directly selecting a terminal or intermediate point of their choice.

3.4 Correspondence Determination

All of the interfaces described above require knowing *correspondences* between the before and after graphs, which are not known a priori. We must determine which nodes in the before graph map to which matching nodes in the after graph, and by extension which nodes in each graph have no match in the other. Using graph structure alone, this problem is hard in the formal sense [12], so we use an approximation technique.

Each node has a distinguished name, and if the nodes are identical in the before and after graphs then their names are the same. The converse, however, is not true: a node name can appear multiple times in a trace, and insertions or deletions can have the same name as existing nodes. We exploit this naming property with a

correspondence approximation approach based on string-edit distance. An overview of the process is shown in Figure 5.

We first serialize each graph through a depth-first search, producing a string of objects. The two graphs shown at left in Figure 5, for example, are transformed into the strings `abcd` and `abce`. In each recursive step, we traverse adjacencies in lexical order so that any reordered nodes from request flow graph collection (e.g., interchanging nodes `b` and `e` in Figure 5(a)) do not produce different strings. This approach is reminiscent of that used in other graph difference comparison heuristics [12].

We then calculate string-edit distance [39] between the resulting strings, with two components considered equal if their names are equal, providing a correspondence between nodes. For example, Figure 5(c) shows three corresponding nodes, one deletion, and one insertion. To obtain these correspondences from the memoization matrix shown in Figure 5(b), at each computation step we maintain a chain of the edits that led to the distance shown at bottom right. We then trace the chain to its end.

Finally, we map the differences computed above onto the graph union of the before and after graphs. Each vertex in the graph union is tagged with one of three types used in displaying the three interfaces — in both graphs, in only the before graph, or in only the after graph. Edges likewise are tagged with one of these three types by comparing the adjacency lists of nodes in the two graphs. The graph union is used to compute the layout of the diff interface, and the three tags control the visualization in each interface (for example, the opacity of nodes in the animation interface).

Of course, this approach is only approximate. Suppose, for example, that the node named `e` in Figure 5 were instead labeled `d`. The resulting serialized strings would both be `abcd`, and no nodes would be considered to be insertions or deletions. We have found, however, that our technique works well in practice on request-flow graphs, in which changes in structure are typically accompanied by changes in node names.

3.5 Common features

All three of our interfaces incorporate some common features, tailored specifically for request-flow graphs. All graphs are drawn with a layered layout based on the technique by Sugiyama et al [38]. This algorithm works well for drawing graphs that can be presented as a hierarchy or a sequence of layers, a property satisfied by request-flow graphs. Layouts that modify this underlying approach enjoy widespread use [11]. Our interfaces

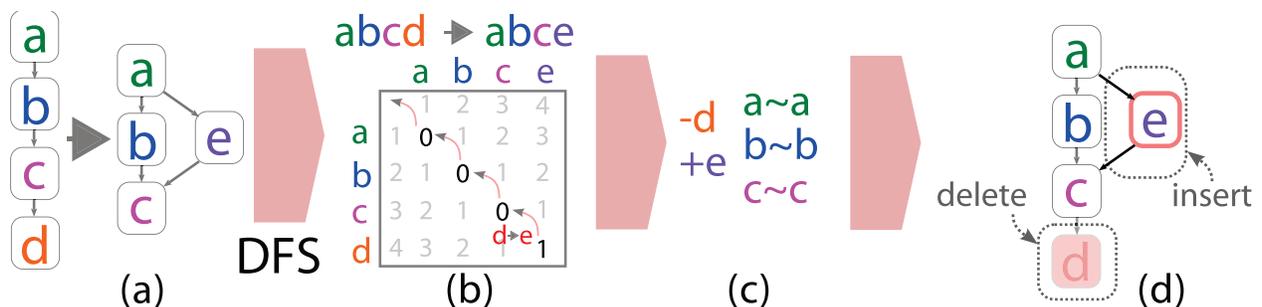


Figure 5: Correspondence determination process. Here we show our method for finding correspondence, for the same synthetic graph as shown in Figure 4. Starting from the before-and-after graph pair shown in (a), we perform a depth-first search to transform the request-flow graphs to strings. These strings are compared by finding their string-edit distance, as illustrated in (b). While computing the string-edit distance, we maintain information about the insertions, deletions, and correspondences between nodes, as shown in (c). Finally, we map these edits back to the original graph structure, as shown in a diff-style format in (d).

use a slightly modified version omitting some stages that are unnecessary for request flow graphs, such as detecting and eliminating cycles (requests cannot move backward in time).

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 considerably smaller than the viewing window.

As mentioned in Section 2, the comparison tool automatically identifies certain edges as having changed statistically significantly between the before and after graphs. The interfaces highlight these edges with a bold red outline.

As drawn in each graph, the length of an edge relates to its latency. Because latencies within a single request-flow graph can vary by orders of magnitude, we do not map latency directly to length; instead, we use a sigmoid-based scaling function that allows both longer and shorter edges to be visible in the same graph.

When graphs contain join points, or locations where multiple parallel paths converge at the same node, a layered layout will likely produce edge lengths that no longer correspond to the values given by the original scaling function. This occurs when one flow completes before another and must wait. Our interfaces illustrate the distinction between *actual* latency and connecting lines by using thinner lines for the latter. An example of this notation appears in the right-hand graph in Figure 4a, between nodes b and c.

4 User study overview & methodology

We evaluated the three interfaces via a between-subjects user study, in which we asked participants to complete five assignments. Each assignment asked participants to find key performance-affecting differences for a before/after request-flow graph pair obtained from Ursa Minor (the distributed system shown in Figure 2). Four of the five assignments used graphs that were the output of request-flow comparison for real problems observed in the system. These problems are described in more detail in Sambasivan et al. [34].

4.1 Participants

Our tool's target users are the developers of the distributed system being diagnosed. Our example tasks come from the Ursa Minor system, so we selected the seven Ursa Minor developers to whom we had access as expert participants. However, due to the limited size of this pool and because we knew some of these participants personally, we also recruited 13 additional non-expert participants. All of these non-experts were generally familiar with distributed systems but not with this system specifically. Although we advertised the study in undergraduate and graduate classes, as well as by posting fliers on and around our campus, all our non-expert participants were graduate students in computer science, electrical and computer engineering, or information networking. The user study took about 1.5 hours and we paid participants \$20.

Potential non-expert participants were required to complete a pre-screening questionnaire that asked about key undergraduate-level distributed systems concepts. To qualify, volunteers were required to indicate that they understood what a request is in the context of a distributed system, along with at least two of five additional concepts: client/server architecture, concurrency and synchronization, remote procedure calls, batching, and critical paths. Of the 33 volunteers who completed the questionnaire, 29 were deemed eligible; we selected the first 13 to respond as participants.

During the user study, each participant was assigned, round-robin, to evaluate one of the three interfaces. Table 1 lists the participants, their demographic information, and the interface they were assigned.

4.2 Creating before/after graphs for the assignments

Our user study contains five total assignments, each requiring participants to identify salient differences between a before/after graph pair. To limit the length of the study and explore the space of possible differences, we removed a few repeated differences from real-problem graphs and added differences of different types. However, we were careful to preserve the inherent complexity of the graphs and the problems they represent. 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 request-flow 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 important differences.

4.3 User study procedure

The study consisted of four parts: training, guided questions, emulation of real diagnoses, and interface comparison. Participants were encouraged to think aloud throughout the study.

ID	Gender	Age	Interface
ES01	M	26	Side-by-side (S)
ES02	M	33	S
ES03	M	38	S
ED04	M	37	Diff (D)
ED05	M	44	D
EA06	M	33	Animation (A)
EA07	M	26	A
7M Avg=34 3S, 2D, 2A			
(a) Participant demographics for experts			
ID	Gender	Age	Interface
NS01	F	23	Side-by-side (S)
NS02	M	21	S
NS03	M	28	S
NS04	M	29	S
ND05	M	35	Diff (D)
ND06	M	22	D
ND07	M	23	D
ND08	M	23	D
ND09	M	25	D
NA10	F	26	Animation (A)
NA11	M	23	A
NA12	M	22	A
NA13	M	23	A
6M, 2F Avg=25 4S, 5D, 4A			
(b) Participant demographics for non-experts			

Table 1: Participant demographics. Our user study consisted of twenty participants, seven of whom were experts (developers of Ursa Minor) and thirteen of whom were non-experts (graduate students familiar with distributed systems). Interfaces were assigned round-robin. The ID encodes whether the participant was an expert (E) or non-expert (N) and the the interface assigned (S=side-by-side, D=diff, A=animation).

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

Table 2: Information about the before/after graph pairs used for the assignments. Assignments 1–3 were used in the guided questions phase (labeled G and described in Section 4.3.2); 4 and 5 were used to emulate real diagnoses (labeled E and described in Section 4.3.3). Four of the five assignments were the output of request-flow comparison for real problems seen in Ursa Minor. The assignments differed in whether they contained statistically significant edge latency changes, other edge latency changes not identified automatically, or groups of structural changes. The graph sizes for the various assignments varied greatly.

4.3.1 Training

In the training phase, participants were shown the Ursa Minor diagram similar to the one in Figure 2. They were not required to understand details about the system, but only that it consists of four components that can communicate with each other over the network. We also presented them with a sample request-flow graph and described the meaning of nodes and edges. Finally, we trained each participant on her assigned interface by showing her a sample before/after graph pair 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 were told that we would be unable to answer further questions after the training portion.

4.3.2 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 (that is, those automatically identified by the request-flow comparison tool as having statistically significant changes in latency distribution).

TASK 2: Find any other edges with latency changes worth investigating. The request-flow comparison tool 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 iterate through the graphs and find edges with notable latency changes that were not highlighted in red.

TASK 3: Find any groups of structural changes. Participants were asked to identify added or deleted nodes in the after graph. To reduce effort, we asked them to identify these changes in contiguous groups, rather than by noting every 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, all of the edge latency changes (statistically significant and otherwise) for the graphs presented in assignment 1 indicate a slowdown in network communication between machines and in write activity within one of Ursa Minor's storage nodes. It is important that participants be able to identify these themes, as doing so 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.

4.3.3 Emulating real diagnoses

In the next phase, participants completed two additional assignments. These assignments, which were less guided than in the prior phase, were intended to emulate how the interfaces might be used in the wild, as 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 which features of the interface they liked or disliked, and to suggest improvements.

4.3.4 Interface preference

Finally, to get a more direct sense of how the interfaces compared, we presented participants with an alternate interface. We asked them to briefly think about the tasks again, considering whether they would be easier or harder to complete with the second interface. We also asked participants which features of both interfaces they liked or disliked. Because our pilot studies suggested the animation interface was most difficult to use, we focused this part of the study on comparing the side-by-side and diff interfaces.

4.4 Scoring criteria

Our user study includes both quantitative and free-form, qualitative tasks. We evaluated participants' responses to the quantitative tasks by comparing them to an "answer key" created by an Ursa Minor developer who had previously used the request-flow-comparison tool to diagnose many of the real problems used to create the assignments. Task 5, which required only a single answer, was scored using accuracy (i.e., does the participant's answer match the answer key?). 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 accurate change out of ten possible ones. In task 3 ("find groups of structural changes"), participants who marked any part of a correct group were given credit for that group.

For task 4 ("identify what the changes represent"), we accepted an answer as correct if it was close to one of several possible explanations, corresponding to different levels of background knowledge. For example, for

one assignment, non-experts would identify the changes as representing extra cache misses in the after graph. Participants with more experience would often correctly identify that the after graph showed a read-modify write, a bane of distributed storage system performance.

We also captured completion times for the quantitative tasks, but found them less useful than precision/recall because times varied greatly based on how sure participants wanted to be of their answers. Some double- and triple-checked their answers before moving on to the next task. Several (usually experts) spent additional time trying to brainstorm *why* the changes had occurred. As a result, we do not present completion times in the results.

We recorded and analyzed participants' comments from each phase as a means to better understand how they approached each assignment, as well as the strengths and weaknesses of each interface.

4.5 Limitations

Our methodology has several limitations. Most importantly, it is difficult to evaluate the true utility of interfaces for helping developers diagnose complex problems without asking them to go through the entire process of debugging a real problem. However, doing so would require a large number of expert participants intimately familiar with the system being diagnosed. As a compromise, we created tasks that tried to tease out whether participants were able to understand the gist of the problem and identify starting points for diagnosis. Even this was sometimes difficult for non-experts. For example, non-experts fared worse than experts on task 4 ("identify what the changes represent"), largely because they were less knowledgeable about distributed systems and Ursa Minor (see Figure 7). Overall, our small sample size limits the generalizability of our quantitative results.

When asked to identify aspects of the interfaces they liked or disliked, several participants mentioned issues with the mechanisms we provided for recording study answers, in addition to issues with the interfaces themselves. This may have affected participants' overall views of the utility of the interfaces.

Finally, many of our participants, especially the non-experts, had a difficult time with the wording of task 1. They often confused "statistically significant latency changes" with "other latency changes worth investigating."

5 User study results

Figure 6 shows the precision/recall and accuracy results for each of the three interfaces. Results for individual tasks, aggregated across all assignments, are shown. Both side-by-side and diff fared well, and their results in most cases are similar for precision, recall, and accuracy. Their results are also similar for the "I am confident my answers were correct" Likert shown in Figure 8. Though animation fared better than or was comparable to the other interfaces for tasks 3, 4, and 5, it fared especially poorly for recall in latency-based tasks (tasks 1 and 2). Participants' comments about animation were also the most negative. Figure 7 shows the same results broken down by participant type. Between experts and non-experts, there was no clear winner between diff and side-by-side. Instead, the choice between them seems to depend on the participant's familiarity with software development as well as the type of task. Animation fares worse than the other two interfaces for experts.

The rest of this section describes key findings from our study. We concentrate our analyses on diff and side-by-side, as these two were the most promising of the interfaces tested.

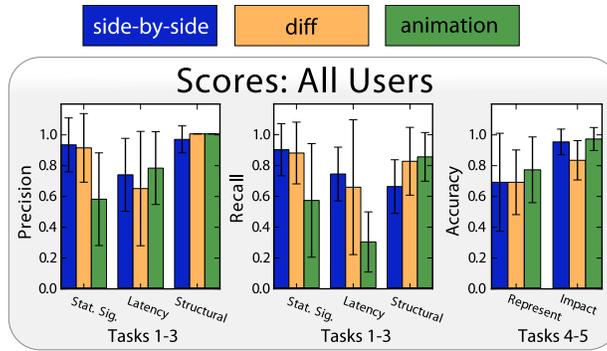


Figure 6: Scoring results. By and large, participants performed reasonably well using both side-by-side (blue) and diff (orange) interfaces. Though animation fared well for certain tasks, such as structural precision, its recall score for latency-based recall tasks was notably poor.

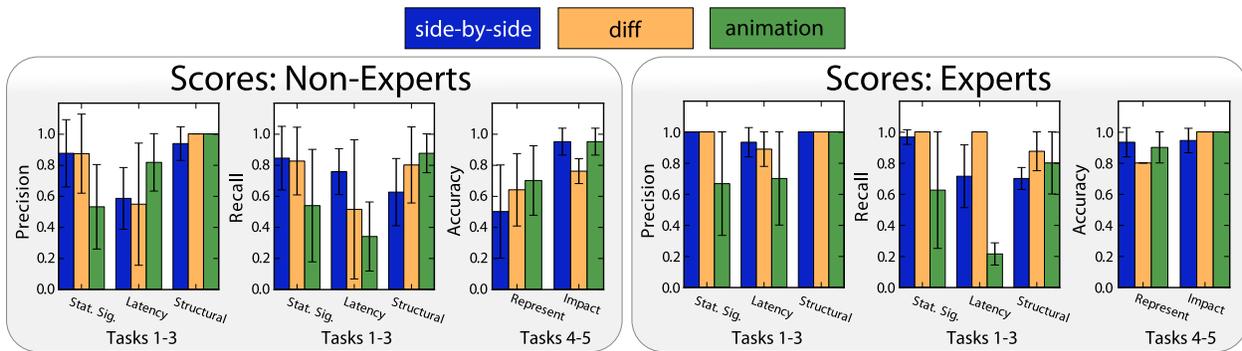


Figure 7: Scoring results by group. As expected, experts’ results are generally better than non-experts. There is no clear winner between diff and side-by-side. Experts fared worse with animation than with the other interfaces.

5.1 Finding differences

For all of the interfaces, we observed that participants found most differences by scrubbing the graphs—i.e., by slowly scrolling downward from the top and scanning for differences. Participants would often zoom in and out during this process in order to obtain a high-level understanding of the graphs. A few participants complained about the amount of scrolling necessary for large graphs, especially those with multiple types of differences (e.g., edge latency and structural). ND05 said, “I do feel like I have to spend a lot of time scrolling up and down here. I feel like there might be a faster way to do that.” NA10 said, “These big graphs make me scared. I have to look [at them] in parts.” Instead of scrubbing, several participants zoomed out to fit the entire graph on the screen when identifying statistically significant differences, as the red highlighting used for these changes was easily distinguishable even when the graphs were small.

When identifying important edge latency differences not marked as statistically significant, users of diff compared lengths of the blue (before) and orange (after) edges. Many users of side-by-side realized they could find such edges by looking for successive correspondence lines that were non-parallel. Users of animation would try to focus on a reference point while repeatedly starting and stopping the animation. Unfortunately, these participants were usually unable to find such points, because all nodes below any difference will move during the animation process. As such, participants found it difficult to find all edge latency changes with this interface, accounting for its low recall score.

When identifying structural differences, users of diff would either look for nodes highlighted in blue or orange

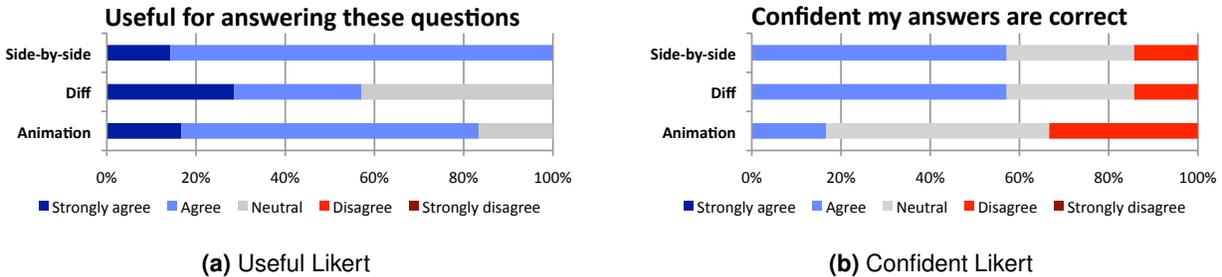


Figure 8: Likert responses, by condition. Each participant was asked to respond to the statements “The interface was useful for answering these questions” and “I am confident my answers are correct.”

or look for nodes with plus or minus signs. Users of side-by-side relied on the lack of correspondence lines between nodes, and users of animation tried to rely on the same process as for edge latency differences.

5.2 Side-by-side is straightforward, but shows too much data

Figure 6 shows that side-by-side and diff’s scores for precision, accuracy, and recall were comparable in most cases, with side-by-side performing slightly better overall. The most notable exception is recall for task 3 (“find groups of structural changes”). Figure 7 shows that side-by-side was comparable to diff for both experts and non-experts in many cases. For recall in task 2 (“find all latency changes worth investigating”), non-experts performed better with side-by-side than diff and the reverse was true for experts. For task 4 (“identify what the changes represent”), non-experts performed worse with side-by-side than the other two interfaces.

Figure 8a shows that participants found side-by-side the most useful interface of the three, as 100% of all participants strongly agreed or agreed to the corresponding Likert question. It was also tied for highest confidence with diff, as shown by Figure 8b.

Participants liked side-by-side because it was the most straightforward of the three—it showed both graphs with correspondence lines clearly showing matching nodes in each. Comparing diff to side-by-side, ND09 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.” ND06 similarly said, “[Side-by-side] gives me a parallel view of the before and after traces because it shows me the lines where one system is heading and the difference between the graphs. . . . The lines are definitely interesting—they help you to distinguish between [the] before and after trace.”

Side-by-side’s simple approach comes at a cost. 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., ES02, ES03, NS01, and NS03). A few participants came up with unique solutions to cope. For example, NS03 gave up trying to identify corresponding nodes between the graphs and 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.

Participants also found it difficult to find differences when one graph was much longer than the other, because correspondence lines would disappear off the top of the screen. This problem was especially prevalent at the end of such graph pairs, because the slope of a correspondence line depends on the sum of all timing changes above it. ES01 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.” ED04 complained that it was more difficult to match up large

changes since the other one could be off the screen. Similar complaints were voiced by other participants (e.g., ES02, NS02).

5.3 Diff is polarizing

Figure 6 shows that the overall precision/recall results for diff and side-by-side are similar, with side-by-side performing only slightly better. However, diff's performance varies more than side-by-side, depending on the type of task. Figure 7 shows that experts performed as well or better with diff than with side-by-side, whereas the reverse was true for non-experts. Figure 8a shows that 57% of participants found diff useful, the lowest percentage for any of the interfaces. However, diff tied side-by-side for confidence, as shown in Figure 8b.

Participants' comments about diff were polarized, with some participants finding it hard to understand and others preferring its compactness. NS04, who fell into the former category, said, "[Side-by-side] may be more helpful than [diff], because this is not so obvious, especially for structural changes." Though participants did not usually make explicit comments about finding diff difficult to use, we found that diff encouraged incorrect mental models in non-expert participants. For example, ND08 and ND09 confused structural differences that resulted in node additions within a single thread of activity with extra parallel activity. It is easy to see why participants might confuse the two, as both are represented by forks, differentiated only by whether there are node deletions that correspond to the additions.

Experts and the more advanced non-experts preferred diff's compactness. For example, ES03 claimed diff's compact representation made it easier for him to draw deductions.

We postulate the results for 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. For example, many developers are familiar with diff tools for text, which would help them understand our graph-based diff technique more easily.

5.4 Diff and side-by-side have contrasting advantages

When users of diff and side-by-side were asked whether they preferred their starting interface or the alternate one presented to them at the end of the study (side-by-side for users of diff and diff for users of side-by-side), almost all participants chose the second, regardless of starting interface. Neither was perfect, and both had advantages when compared to the other. ED04 was dismayed when we asked him which one he preferred: "If had to choose between one and the other without being able to flip, I would be sad."

Table 3 summarizes the comparative advantages of both interfaces, drawn from direct comparisons as well as our observation of how participants used them. Side-by-side's main advantages have to do with its straightforwardness and simplicity of representation, as detailed in Section 5.2. Forks in the graphs always represent extra parallel activity, and changes in the slope of correspondence lines represent edge latency changes. Participants also liked that it used horizontal screen space better than diff, which only shows one (usually) long, skinny graph at a time. Diff's main advantages have to do with its compactness and lack of clutter, as detailed in Section 5.3. It is also easier to use to find small edge latency differences, as the items being compared are much closer to one another (adjacent blue and orange edges vs. correspondence lines). As a result of these relative strengths, we believe diff is more useful for large graphs and for experts. In contrast, side-by-side is preferred for smaller graphs and novices.

Side-By-Side	Diff
<ul style="list-style-type: none"> ✦ Simple Presentation of two graphs simultaneously is conceptually straightforward ✦ Unambiguous Graph forks always represent parallelism, never structural changes ✦ Slope=Latency Changes in slope of correspondence lines indicate changes in latency ✦ Lines=Structure Lack of correspondence lines indicate structural changes ✦ Space-Filling Makes better use of horizontal screen space 	<ul style="list-style-type: none"> ✦ Compact Limits clutter: combines information for comparison, uses fewer lines, and lines do not overlap ✦ Annotated Explicit encoding with plus and minus symbols helps spot structural changes ✦ Skew-Free Corresponding nodes remain horizontally aligned ✦ Latency is Evident Differences in orange and blue line lengths indicate latency changes clearly

Table 3: Advantages of side-by-side and diff. This table shows the relative benefits of the side-by-side and diff interfaces, as drawn from participants' behaviors and comments.

5.5 Animation has clear weaknesses

Figure 6 shows that though animation's performance on many tasks was comparable to the other interfaces, its performance on latency-based recall tasks (2 and 3) was very poor. Figure 7 shows that experts fared worse with animation than the other two interfaces. Figure 8a shows that 83% of participants found animation useful, a curiosity because participants' comments about animation were the most negative. However, Figure 8b shows only 17% of participants were confident in their answers.

With animation, all differences between the two graphs appear and disappear at the same time. This cacophony confused participants, especially when multiple types of differences were present. In such cases, edge latency changes would cause existing nodes to move down and, as they were doing so, trample over nodes that were fading in or out due to structural changes. EA07 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." NA11 explicitly told us that the fact the graph moved so much annoyed him.

During the animation process, all nodes below a difference will move. This frustrated participants, as they were unable to identify static reference points for determining how a graph's structure changed around a particular node or how much a given edge's latency changed. NA10 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." NA12 said he didn't like animation 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."

A final negative aspect of this interface is that it implies the existence of a false intermediate state between the before and after graphs. As a result, NA13 interpreted the animation as a timeline of changes and listed this as a feature he really liked.

5.6 Automation results must match users' expectations

One surprising aspect of our study was participants' difficulty in answering task 1, which asked them to mark all edges with statistically significant latency changes. Since these edges were automatically highlighted in

red, we anticipated participants would have no difficulty in finding them. However, most of our participants did not have a strong background in statistics, and so they took “statistically significant” to mean “large changes in latency,” generating much confusion and accounting for the lower than expected scores. In reality, even large differences in average latency may not be statistically significant if the variance in individual latency values is very high. Conversely, small differences may be statistically significant if variance is low. Request-flow comparison uses statistical significance as the bar for automatically identifying differences because it bounds the expected number of false positives when there are large latency increases.

As a result of their incorrect mental model, some participants (usually non-experts) failed to differentiate between task 1 and task 2, the latter of which asked participants to find other (non statistically-significant) edges with latency changes worth investigating. Participants were especially concerned with why small changes in latency were labeled statistically significant. NA10 complained, “I don’t know what you mean by statistically significant—maybe it’s statistically significant to me,” and followed up with “I am paying special attention to the ones that are marked in red, but they don’t seem to be changing. . . it’s very confusing.” Confusion about the red highlighting affected results of other tasks as well, with some participants refusing to mark a change as having the most impact unless it was highlighted.

We could have worded task 1 better to avoid some of this confusion, but these results demonstrate the important point that the results of 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 visualization techniques and more advanced training may be needed to achieve this.

6 Design lessons

Moving forward, the insights we gleaned from evaluating the three interfaces suggest a number of directions for improving diagnosis visualization. Here we highlight a few key common themes.

A primary recurring theme is the need to selectively reduce the complexity of large diagnosis outputs. Even when they were navigable, graphs with hundreds of nodes pose an obstacle to understanding the output at a high level. In our interface, zooming in and out at the image level allows the entire comparison to be seen, but it does not provide intuition into the meaning of the graph as a whole. A number of possible concrete solutions could help alleviate this difficulty for request-flow comparison — for instance, coalescing portions of the comparison that are the same in both graphs, or grouping sequences of similar operations (mentioned by ND09, ES02, ED04, and ES01). A small context view indicating the navigation position in the graph structure would also help (EA07, ED05).

When displaying large and complex graphs, providing anchor points for analysis is also important. This theme was evidenced by the struggle many users had with the increasing skew in the side-by-side and animation layouts, as well as the inability to quickly trace a correspondence from one graph to another (e.g., ES02 and NA10). A future interface could realign graphs or allow users to anchor the comparison around a selected point.

Ensuring that automation behaves predictably is critical to avoiding the confusion mentioned in Section 5.6. False positives presented by an interface lead users to question their own intuition and the criteria used for automatically directing their attention to portions of an output. On the other hand, false negatives lead users to question the effectiveness of the automated system at pinpointing issues. Ideally, automated tools would produce perfect output, but failing that, this issue can be mitigated by providing users more training and

intuition for the failure modes of automation. Offering users control over the parameters that control the output of automation can also help.

Several participants struggled with comparing the relative impact of changes that were far apart in the graph. Three of our experts (EA07, ES02, ED05) suggested labeling each edge with its latency. Others requested a way to measure the total latency for one chunk of a graph. Both of these features are included in an expanded version of our visualization tool; we presented our participants with the simplified version in order to focus on how the different visual elements of the three interfaces affected their success.

7 Related work

We briefly survey related work on the topics of finding correspondences between graphs, visualizing differences between graphs, the effectiveness of animation, and visualization for system diagnosis generally.

Graph difference techniques: A variety of algorithms have been proposed to find the difference or edit distance between two graphs with unknown correspondence. We direct the interested reader to [12] for a survey. As finding graph correspondence in the general case is hard, these algorithms are limited in applicability [40, 41], approximate [2, 10, 22], or both. Few approaches have a theoretical foundation for correctness [12, 33], and we similarly make no attempt to provide a formal model for our technique.

Visual graph comparison: Given two graphs with a known correspondence, visual analysis techniques have been proposed to help users understand differences and similarities. Different types of graphs and features to compare have merited different approaches. For trees, TreeJuxtaposer [27] analyzes structural changes in large graphs, and TreeVersity [16] addresses both structural and node-value changes on smaller data. G-PARE [35] analyzes only value changes on general graphs. Visualizing sequences of graphs [3, 17] is one of few domains where user studies have been employed to gauge the effectiveness of different approaches. In particular, one user study compares four approaches to sequence visualization, finding that a “difference map” (roughly akin to the diff view for unweighted, undirected graphs) was significantly preferred but not more useful.

Animation: The effectiveness of animation to help users analyze data (e.g., identify trends or differences) is controversial, with some studies finding it more useful than static approaches (e.g., small multiples) and others finding it less useful. We refer readers to Robertson et al. [32] for a summary of these studies. The lack of agreement suggests that animation’s effectiveness may be domain specific. For example, Archambault et al. suggest that animation may be more effective for helping users understand the evolution of nodes and edges in a graph whereas small multiples may be more useful for tasks that require users to read node or edge labels [4]. Robertson et al. compare animation’s effectiveness to that of small multiples and one other static approach for identifying trends in Gapminder Trendalyzer [13] visualizations. They find that animation is more effective and engaging for presenting trends, whereas static approaches are more effective for helping users identify them.

Visualization for system diagnosis: Compared to the wealth of recent work in automated diagnosis, there have been relatively few efforts investigating effective visual methods for understanding the results. For instance, the Dapper tracing infrastructure at Google focuses on providing APIs to build bespoke tools, with little investigation or evaluation of its default interface [37]. Indeed, a recent survey of important directions for log analysis concludes that because humans will remain in the analysis loop, visualization research is an important next step [29].

One project in this vein is NetClinic, which considers root-cause diagnosis of network faults [23]. The authors find that visualization in conjunction with automated analysis [19] is helpful for diagnosis. As in this study, the

tool uses automated processes to direct users' attention, and the authors observe that automation failures inhibit users' understanding. In another system targeted at network diagnosis, Mansmann et al. observe that automated tools alone are limited in utility without effective presentation of results [26]. Like many other network monitoring efforts, however, the proposed solution primarily focuses on improving display of the underlying data rather than the output of an automated tool.

8 Summary

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 describes a 20-person user study comparing three interfaces for presenting the results of request-flow comparison, one particular automated problem localization technique. Via quantitative analyses and qualitative statements from users, we found two of the three interfaces to be effective and identified design guidelines for further development. We believe these guidelines can be applied more broadly to visualizing many automated tools for building and evaluating computer systems.

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 *Proceedings of the 4th USENIX Conference on File and Storage Technologies, FAST '05*. USENIX Association, Dec. 2005. 2
- [2] K. Andrews, M. Wohlfahrt, and G. Wurzinger. Visual Graph Comparison. In *Proceedings of the 13th International Conference on Information Visualisation, IV '09*, pages 62–67. IEEE, Jul. 2009. 17
- [3] D. Archambault. Structural differences between two graphs through hierarchies. In *Proceedings of Graphics Interface 2009, GI '09*, pages 87–94. Canadian Information Processing Society, 2009. 17
- [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. 17
- [5] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '07*, pages 13–24. ACM, Aug. 2007. 1
- [6] N. G. Belmonte. The Javascript Infovis Toolkit. <http://www.thejit.org>. 4
- [7] 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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, pages 15–29. USENIX Association, Nov. 2006. 3
- [8] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation, NSDI'04*, pages 23–36. USENIX Association, Mar. 2004. 1
- [9] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles, SOSP '05*, pages 105–118. ACM, Oct. 2005. 1
- [10] J. Eder and K. Wiggisser. Change detection in ontologies using DAG comparison. In *Proceedings of the 19th International Conference on Advanced Information Systems Engineering, CAiSE'07*, pages 21–35. Springer-Verlag, 2007. 17

- [11] 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*, Mutzel, M. Junger, editor, pages 127–148. Springer-Verlag, Berlin, 2003. 2, 6
- [12] 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. 5, 6, 17
- [13] Gapminder. <http://www.gapminder.org>. 17
- [14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03. ACM, Dec. 2003. 3
- [15] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4):1–29, Oct. 2011. 4
- [16] J. A. G. Gomez, C. Plaisant, B. Shneiderman, and A. Buck-Coleman. Interactive visualizations for comparing two trees with structure and node value changes. Technical Report HCIL-2011-22, University of Maryland, 2011. 17
- [17] C. Görg, P. Birke, M. Pohl, and S. Diehl. Dynamic graph drawing of sequences of orthogonal and hierarchical graphs. In *Proceedings of the 12th International Conference on Graph Drawing*, GD '04, pages 228–238. Springer-Verlag, 2004. 17
- [18] A. Inselberg. *Parallel coordinates: visual multidimensional geometry and its applications*. Springer-Verlag, Sep. 2009. 4
- [19] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '09, pages 243–254. ACM, Aug. 2009. 1, 3, 17
- [20] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST '10, pages 4–17. USENIX Association, Feb. 2010. 1
- [21] S. P. Kavulya, S. Daniels, K. Joshi, M. Hultunen, R. Gandhi, and P. Narasimhan. Draco: statistical diagnosis of chronic problems in distributed systems. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN'12. IEEE, Jun. 2012. 1
- [22] I. Lin and S. Kung. Coding and comparison of DAG's as a novel neural structure with applications to on-line handwriting recognition. *IEEE Transactions on Signal Processing*, 45(11):2701–2708, Nov. 1997. 17
- [23] Z. Liu, B. Lee, S. Kandula, and R. Mahajan. NetClinic: interactive visualization to enhance automated fault diagnosis in enterprise networks. In *IEEE Symposium on Visual Analytics Science and Technology*, VAST '10, pages 131–138. IEEE, Oct. 2010. 2, 17
- [24] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and merging files with GNU diff and patch*. Network Theory Ltd, 2002. 4
- [25] A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao. Towards automated performance diagnosis in a large IPTV network. In *Proceedings of the 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '09, pages 231–242, Aug. 2009. 1
- [26] 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 *Proceedings of the Symposium on Computer Human Interaction for the Management of Information Technology*, CHI+IT '09, pages 3:19–3:28. ACM, 2009. 1, 2, 18
- [27] T. Munzner and S. Tasiran. TreeJuxtaposer: scalable tree comparison using Focus+Context with guaranteed visibility. *ACM Transactions on Graphics*, 22(3):453–462, 2003. 17
- [28] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12. USENIX Association, Apr. 2012. 3
- [29] A. J. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *ACM Queue*, 9(12):30:30—30:40, Dec. 2011. 1, 17

- [30] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, pages 191–200. IEEE, Apr. 2010. 1
- [31] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '06, pages 115–128. USENIX Association, 2006. 1
- [32] G. Robertson, R. Fernandez, D. Fisher, B. Lee, and J. Stasko. Effectiveness of animation in trend visualization. In *IEEE Transactions on Visualization and Computer Graphics*, INFOVIS '08, pages 1325–1332, Nov. 2008. 17
- [33] A. Robles-Kelly and E. R. Hancock. Graph edit distance from spectral seriation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(3):365–78, Mar. 2005. 17
- [34] R. R. Sambasivan, A. X. Zheng, M. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, pages 43–56. USENIX Association, Mar. 2011. 1, 2, 3, 7
- [35] H. Sharara, A. Sopan, G. Namata, L. Getoor, and L. Singh. G-PARE: a visual analytic tool for comparative analysis of uncertain graphs. In *Proceedings of the IEEE Conference on Visual Analytics Science and Technology*, VAST '11, pages 61–70. IEEE, Oct. 2011. 17
- [36] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly detection. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 85–96. ACM, Apr. 2009. 1
- [37] 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. 17
- [38] 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, 1981. 6
- [39] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, Jan. 1974. 6
- [40] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *Society for Industrial and Applied Mathematics Journal on Scientific Computing*, 18(6):1245–1262, 1989. 17
- [41] K. Zhang, J. T. L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407. Springer-Verlag, Jul. 1995. 17