Tracey - Distributed Trace Comparison and Aggregation using NLP techniques

Vaastav Anand
vaastav@cs.ubc.ca

Joseph Wonsil
jwonsil@cs.ubc.ca

Abstract

Distributed systems are widespread in usage. Yet, they continue to be marred by bugs. Distributed tracing is a widely adopted approach that gives engineers visibility into cloud systems. Existing tracing tools support analysis of a *single* request and are most useful for debugging correctness issues.

However, diagnosing an issue in the processing of a request, requires comparing the execution of a buggy request to a non-buggy request or even an aggregate set of requests. Some issues even require comparing the behavior of two different sets of requests to identify a potential issue. Existing trace analysis tools either do not support these use cases or produce an output that is not understandable.

To rectify this, in this paper we propose a new approach for performing trace comparison and aggregation. The key insight of our approach is to derive a text representation for each trace and then perform aggregation and comparison on texts. The benefit of this approach is twofold: we can leverage text summarization and comparison algorithms; the output produced is text which is understandable for users. We present algorithms for generating a text representation from a trace, summarization of these text representations to generate a summary of traces, and comparison of traces.

1 Introduction

Distributed systems are prevalent in society to the extent that billions of people either directly or indirectly depend on the correct functioning of a distributed system. From banking applications to social networks, from large-scale data analytics to online video streaming, from web searches to cryptocurrencies, most of the successful computing applications of today are powered by distributed systems. The meteoric rise of cloud computing in the past decade has only increased our dependence on these distributed systems in our lives.

Tasks like monitoring, root cause analysis, performance comprehension require techniques that cut across component, system, and machine boundaries to collect, correlate, and integrate data. In the past decade, distributed tracing has emerged as an effective way to gain visibility across distributed systems (Mace et al.; Mace and Fonseca; Fonseca et al.). Today distributed tracing frameworks are deployed at all major internet companies (Kaldor et al.; Sigelman et al., 2010; Blog, 2018); notable open-source examples include OpenTelemetry (Flanders, 2019), Jaeger (Jaeger), and Zipkin (Twitter); and observability-focused companies offer platforms centered on analysis of traces (LightStep).

Distributed tracing tools arose out of a need to understand the behavior of *individual requests*: identifying the specific services invoked by a request, diagnosing problematic requests, and debugging correctness issues (Fonseca et al.; Sigelman et al., 2010; Mace). As a result, each trace only tells the story of a single request. A trace represents the path of one request through the system and contains information such as the timing of requests, the events executed, and the nodes where these events were executed. Moreover, traces can be used to identify slow requests and understand the difference between request executions.

However, as distributed tracing is designed for production distributed systems, a large volume of data is produced on a daily basis. It is humanly impossible to manually analyze each trace and draw inference about the system as a collective. Current analysis tools primarily focus on visualizing a single trace and provide little help to the user for analyzing a large amount of data. Moreover, current analysis tools struggle with highlighting the difference between any given two traces due

to the complex temporal and structural properties of traces. Both of these limitations results in the lack of techniques for comparing a trace with an aggregate set of traces to explain to a user how a potentially erroneous trace might differ from a set of pre-identified "good" traces.

In this paper, we present techniques for comparing two traces, aggregating a set of traces, and comparing a trace with a set of traces. The key idea behind our techniques is to first convert each trace into an equivalent text representation and then construct techniques for performing aggregation and comparsion. The benefits of such an approach are two-fold: (i) There has been a plethora of research in the fields of text comparison and text summarization and we can leverage these existing techniques; (ii) Language is a proven way of communicating complex information in an understandable format for human beings.

To this extent, we describe a novel framework that generates a text representation for each trace detailing the execution behavior of the system for the particular request. We then model the trace comparison tasks as text similarity tasks and present a metric for calculating the difference between the two traces based on the edit distance between their corresponding text representations. Lastly, we present trace aggregation as a multi-document summarization task where each trace corresponds to a single document. Section 2 describes the trace data. Section 3 describes the user-tasks and how we model them as NLP tasks in detail. Section 4 explains the design, implementation, and technical details of our techniques. Section 5 presents an evaluation of our techniques and Section 6 discusses the limitations of our techniques and how we plan to address them in future work.

2 Data

2.1 Trace Structure

A trace is a Directed Acyclic Graph (DAG) of spans. A span can be thought of as a particular task that a system performs to execute a request. The granularity of the task is user-defined and controlled. A span can represent anything from a single function execution, a single thread execution, or a single operating system process comprised of multiple threads. Spans are connected to each other by parent-child relationships. Each span records its timing and duration, as well as arbitrary key-value annotations provided by a developer: such as logging a span's arguments. Within a span, developers can also add *events*, which are typically **unstructured, human-annotated slog messages.** Span annotations and events are developer-defined and vary from system to system. Each individual trace can be very large, comprising thousands of spans and events (Kaldor et al.; Las-Casas et al., 2019; Flanders and Shkuro, 2019), and production systems capture traces for millions of requests per day (Kaldor et al.).

Most distributed tracing frameworks represent traces using spans (Jaeger; Flanders, 2019; Sigelman et al., 2010), but some frameworks are based only on events (Fonseca et al.). For event-based frameworks, it is straightforward to group events together into spans (e.g. events occurring in the same thread). In this paper we use the term *task* to refer to both of these concepts. In span-based tracing frameworks a task corresponds to a span and in event-based tracing frameworks a task corresponds to a collection of events.

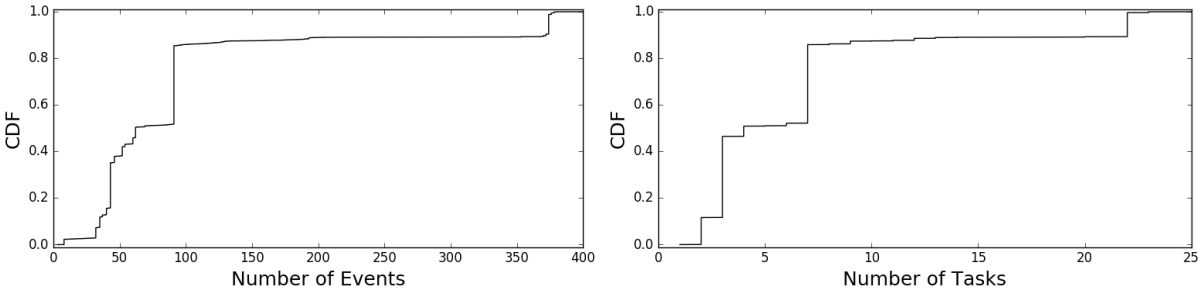
2.2 Dataset

For the development and evaluation of our techniques, we use the open source deathstarbench trace dataset (Anand and Mace, 2019). The dataset contains 22285 individual traces obtained from the DeathStarBench open-source benchmark for cloud microservices (Gan et al.). The captured traces are from 7 different API types (Register user, Follow user, Unfollow user, Composepost, Write timeline, Read timeline, Read user timeline). Internally,the benchmark comprises 36 microservices; each high-level API call invokes an overlapping subset of the services. In addition to datasets of regular workloads, the dataset also contains two types of anomalous traces: one with manually triggered exceptions in the internal microservices; and one arising accidentally from a configuration error in the deployment causing docker containers to intermittently restart and services to be temporarily unavailable.Figure 1 shows the CDF of the number of events and the number of tasks for the dataset.

3 User Tasks

In this section, we explain the user tasks that we want to accomplish and how we model these user tasks as traditional NLP tasks. The list of tasks are as follows:

1. Creating a text representation for any given trace. We model this task as a *Text Generation*



(a) (b)

Figure 1: (a) CDF of the number of events per trace; (b) CDF of the number of tasks per trace; in the DeathStarBench trace dataset

task that generates text from a data source, in this case, tracing data. Thus, this is a Data2Text task.

2. Generating an aggregate representation of traces. We model this task as a *Multi-Document Summarization* task where each document is the text representation generated for a task from a single trace.

3. Comparing and explaining the differences between two traces. We model this task as a *Text Similarity* task where instead of comparing the raw structure of the two traces, we instead compare their corresponding text representations. The difference between the two traces can then be modeled as a function of the edit distance between their corresponding text representations.

4. Comparing one trace to an aggregate set of traces. We also model this task as a *Text Similarity* task where we compare the text representation of a trace with the generated summary of the aggregate set of traces using our technique described in Task 2. Similar to Task 3, the difference between the trace and the aggregated summary can be modeled as a function of the edit distance between their corresponding text representations.

4 Contribution

4.1 Trace2Text

The task of text generation from data can be divided into three modular interdependent tasks - (i) *content planning* defines which parts of the input fields or meaning representations should be selected; (ii) *sentence planning* determines which selected fields are to be dealt with in each output sentence; and (iii) *surface realization* generates those sentences. To obtain processed trace data, we use a backend server for trace processing that we had built as part of the visualization course last semester[1]. The input data for each trace includes, some overview information of the trace, the list of events, and the list of tasks for that trace. For each event, the human annotated-text as well as the probability of that event occurring are included. For each task, a list of concurrent tasks that were happening during the task's execution are also included. *The goal is to generate a text representation of the trace that includes information describing the execution of the trace and overview information that would be useful for the user.*

```
① The trace was created on 31 May, 2019 and it
    ↪ took around 422 seconds to complete.
② The trace was associated with the following
    ↪ tags: ComposePost NginxWebServer5.
③ The trace had 388 events, out of which 28
    ↪ events had less than 25.0 chance of
    ↪ occurring.
④ The execution of the request was performed
    ↪ by 22 tasks. 3 tasks had latency that
    ↪ ranked higher than the 95th percentile of
    ↪ the latency distribution for the respective
    ↪ task.
⑤ Task performing the operation
    ↪ UrlShortenHandler::
    ↪ UploadUrlsMongoInsert had the
    ↪ maximum amount of contention with 7
    ↪ other tasks performing the same
    ↪ operation at the same time for different
    ↪ requests.
```

[1]The source code of the backend server is located at `https://github.com/vaastav/TraViz/tree/master/traviz_backend`

Figure 2: Annotated overview paragraph generated for a trace

Content Planning The content planner parses the input data to extract the information needed to generate the overview text and the execution text. From the list of events, the number of anomalous events are extracted. An anomalous event is an event that has probability lower than a pre-defined threshold. The content planner also constructs a DAG of the events based on the causal relationships of the events. For each task in the list of tasks, an array of temporally ordered events are created that occurred on that task. The task with the most number of concurrent tasks is also extracted.

Sentence Planning The sentence planner separates the content by planning to generate one paragraph for each task in the trace as well as one paragraph for overview information about the trace. The information contained inside the paragraph would be the name of the task that created this task and labels from the events associated with the task. The overview paragraph provides information about the trace such as latency, tags, and other metadata. It also includes information about anomalous events and tasks.

Surface Realization The surface realization is currently static as it is done based on a pre-defined template. Figure 2 shows an annotated example of the overview paragraph generated for a trace. ① provides information about when the trace was created and what was the latency of the task. ② mentions the tags associated with the trace. ③ contains the total number of events in the trace and the number of anomalous events. ④ has the number of tasks in the trace and the number of tasks which have latencies that lie in the 95th percentile of the latency distributions of their respective tasks. ⑤ provides the name of the task that had the most number of concurrent tasks executing at the same time. Figure 3 shows an annotated example of the paragraph generated for a task in a trace. ① describes which task, if any, created this particular task. ② is the name of the task. Each sentence following ② is the human-annotated label of the corresponding event that occurred on that task. The sentences are ordered w.r.t to the ordering of the events for that task.

① Task NginxWebServer created task
↪ MediaHandler::UploadMedia.
② MediaHandler::UploadMedia.
Uploading media to compose post service.
↪ Popping client from client pool.
↪ Obtaining lock on client pool mutex.
↪ Obtained lock on client pool mutex. No
↪ client available in client pool. Creating a
↪ new client. Releasing lock on client pool
↪ mutex. Connecting to client. Pushing a
↪ client into client pool. Acquiring lock on
↪ mutex. Acquired lock on mutex. Pushing
↪ client back into client pool. Releasing
↪ lock on mutex. MediaHandler::
↪ UploadMedia complete.

Figure 3: Annotated paragraph generated for a task in a trace

4.2 Trace Summarization

We do trace summarization at the granularity of a task. Thus, instead of summarizing the text representations of all the traces we want to summarize, we instead summarize the text representations for each task across all traces. If a set of traces has 30 different tasks, we perform 30 different multi-document summarizations to obtain a summary for each task. These task-specific summaries are then concatenated together to form the summary of the traces. The reason behind doing summarization at the granularity of a task is to ensure that there is no cross-contamination of information between the tasks as we don't want the summarized result to have an incorrect text representation. Additionally, the format of that summary will now match the overview generated for a single trace. This allows us to then be able to make clean comparisons between a single trace and a group of traces using their summary. Since each task has a relatively set behavior, we can infer that the overview paragraphs describing it will never changes drastically during normal execution. Even though it may be repeated hundreds of times, there will not be an explosion in the number of unique sentences found across traces. This means that the summarization of a task does not have to be a complex operation. We break down our summarization in three steps - Preprocessing, Graph Construction, and Text Conversion - that we describe below.

Preprocessing In the preprocessing step, text representation for each trace is broken into text representations of each task. Based on our text generation algorithm, text representation for a task in a trace is a paragraph. Text representations from multiple traces are then grouped together by task.

For each task, there are multiple paragraphs that need to be summarized. Each such paragraph is represented as its own document such that there are multiple documents available for each task.

Graph Construction For each task, we construct an aggregate weighted graph from the documents associated with that task. Each node in the graph represents one unique sentence across the documents. Uniqueness is defined not only by the content of the sentence but also by the prefix of sentences that were before the sentence. This is done to ensure that when the documents are being aggregated, two sentences are only merged together if they have the same preceding sentences. Each edge represents the causal and temporal ordering between sentences. This is constructed by adding an edge between each pair of adjacent sentences. The weight of the edge represents the number of documents in which that edge was seen.

Text Conversion To convert the graph into text we first create a topological sort ordering of the vertices in the graph. This is to flatten the graph structure as the aggregation would have induced branches in the graph because of deviations. We choose a topological sort ordering as it guarantees to preserve the order between sentences from all the documents. Once we have the ordering, we simply concatenate the labels from the nodes according to the ordering to generate the summary for that particular task. Currently, we don't use the weights of the edges in the text generation.

4.3 Trace Diff

To compare two traces, or one trace and one aggregate set of traces, or two different aggregate sets of traces, we choose to compare their corresponding texts. For calculating the difference between the two text representations, we use Google's diff-match-patch library[2]. For explaining our algorithm, we choose to explain the comparison of two traces but the same algorithm applies for the other two comparison scenarios. However, instead of just computing the diff between the two texts, we strive for a much finer granularity, and instead compute the diff between the texts by computing the diffs between the common tasks amongst the two traces. Tasks that are only present in either trace are automatically treated as insertions or deletions in the diff. Figure 4 shows the output of the diff function applied to one common task in two traces. The uncolored, normal text represents sentences (i.e. events) that are present in both traces. The struck-out, red text represents the text that is only present in trace 1 but not in trace 2. The italic, green text represents the text that is only present in trace 2 but not in trace 1.

Task NginxWebServer created task
↪ MediaHandler::UploadMedia.
↪ MediaHandler::UploadMedia.
↪ Uploading media to compose post
↪ service. Popping client from client
↪ pool. Obtaining lock on client pool
↪ mutex. Obtained lock on client pool
↪ mutex. ~~No client available in client pool. Creating a new client~~*Popping client from front of the pool*. Releasing lock on client pool
↪ mutex. Connecting to client. Pushing
↪ a client into client pool. Acquiring
↪ lock on mutex. Acquired lock on
↪ mutex. Pushing client back into
↪ client pool. Releasing lock on mutex.
↪ MediaHandler::UploadMedia
↪ complete

Figure 4: Text Difference for a task from 2 traces

Trace Distance Based on our diff function above, we also provide a new metric for calculating the distance between any two traces. Let T_1 be the set of tasks in Trace 1 and T_2 be the set of tasks in Trace 2. Then, the distance between the two traces is the sum of the distances for each task in T_1 and T_2. Equation 1 shows the formal definition of the function for calculating the distance between the text representations of two traces. The disttask function for a task defines the distance calculated for this task between Trace 1 and Trace 2. If the task is present in both the traces, then the cost is simply the Levenshtein distance between the paragraphs for this task in the two traces. However, if the task is only present in one of the traces, then the distance is the number of sentences in the paragraph for that task multiplied by some pre-defined task missing penalty, P. Currently, P is set to 10 but can be changed depending on the needs of the users. Equation 2 shows the formal definition for calculating the distance between the text representations of a task in two traces.

[2]Library available at `https://github.com/google/diff-match-patch`

$$\texttt{distance} = \sum_{t \in T_1 \cup T_2} disttask(t) \tag{1}$$

$$disttask(t) = \begin{cases} \texttt{Levenshtein}(t1, t2), \texttt{if } t \in T_1 \cap T_2, t1 \in T_1, t2 \in T_2 \\ P * \texttt{numsentences}(t), \texttt{if } t \in (T_1 \setminus T_2) \cup (T_2 \setminus T_1). \end{cases} \tag{2}$$

5 Evaluation

5.1 Experimental Setup

We perform a quantitative evaluation to evaluate the scalability of our techniques as well as the quality of the text, comparisons, and the summaries generated. For the Text Generation and Trace Comparison evaluation, all results were collected on an Intel i7-core 3.1GHz processor machine with 32GB of RAM. For the Trace Summarization evaluation, all results were collected on an Intel(R) Xeon (R) CPU E5-2690 v3 @ 2.60GHz with 56GB RAM and an NVIDIA GK210GL [Tesla K80] GPU.

In addition to the quantitative evaluation, we also performed a qualitative evaluation by conducting an informal user study with 1 user who is one of the leading experts in distributed tracing. Due to time restrictions, the user was only able to provide feedback about the text generation and trace comparison techniques.

5.2 Text Generation

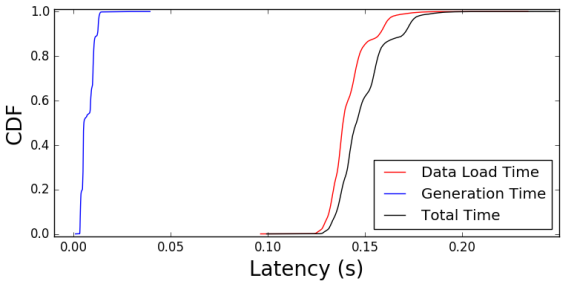


Figure 5: CDF of generating text for all the traces in the DeathStarBench dataset

Quantiative Analysis To measure the efficiency of our text generation approach, we measure the time taken to generate the text for every trace in the DeathStarBench dataset. Figure 5 shows the CDF of the breakdown of the total time taken to generate the text. The time taken to generate the text is dominated by the time taken to load the data from the backend server. However, once the data is available, the time taken to generate the text is less than 10 milliseconds for all traces.

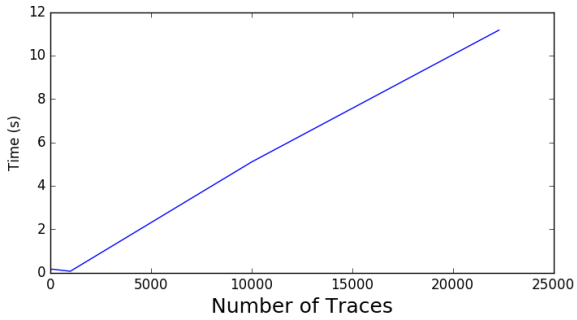


Figure 6: Time taken for preprocessing traces for summarization

Qualitative Analysis To evaluate the quality of the text generated, the user mentioned the following:"The first sentence is the interesting one here; the rest of the sentences are a bit difficult to parse. The interesting parts of the first sentence are the comparisons to general statistics about the trace dataset. I like the last sentence, because it's starting to push towards deriving a root cause for the latency (ie, contention with other tasks), and I think when it's presented as text that's a very digestable representation (vs. some sort of visual interface)." This suggests that the expert user believes that the overview paragraph is very useful for explaining root causes of potential problems and does so better than a potential visual interface might. Although, we can still improve our overview by performing comparison for more statistics that a user might care about. Additionally, the expert user also suggested "to prune away any boring information and try to get at any root causes (or simply say that the request was normal)". This suggests that text that the execution information being shown to the user is too verbose and we might need to infer some high-level information about the trace instead.

5.3 Trace Summarization

We explored three metrics to evaluate the summarization of traces. First to determine how reasonable our summarization method is we performed a scalability microbenchmark using a single task from multiple traces. Then we measure the quality

# of Docs	OS	Potara
5	0.103	77.429
10	0.105	160.240
25	0.107	404.716
100	0.109	1711.283
500	0.140	18118.918

Table 1: Time (in s) taken by Potara and Our Summarizer (OS) to summarize varying number of documents for a task

Num Traces	OS	Potara
5	0.109	31.401
10	0.116	43.979
25	0.120	625.489
100	0.156	6096.836
1000	0.483	72176.924
10000	3.851	DNF
22290	8.224	DNF

Table 2: Time taken (in s) by Our Summarizer (OS) and Potara to summarize a varying number of traces

of the summary based on the number of unique sentences it was able to capture from the original documents. Finally, we perform a macro-scalability test by summarizing all tasks from multiple traces. For each metric we run our own algorithm as well as the potara[3] summarization tool as a baseline. We chose this tool as this was the only open-source multi-document summarizer that we could find that would run out of the box on our data. We also tried using BERT but could not properly configure it to generate meaningful summaries. In addition we also measured how long does it take to perform the preprocessing step. Figure 6 shows that the preprocessing time grows linearly with the number of traces.

Micro-scalability We tested the scalability of the summarizers at a document-level. We split up the summarization to be at a task level. This means that each task in a given summarization job will have its own set of documents. To perform a micro-scalability evaluation we looked at only a single task's documents. For this benchmark, we chose a task that appeared in a large number of traces so that we can get a better estimate about the scalability of summarizing a large number of documents for a task. We ran the summarization algorithm on an increasing number of documents for a particular task. Table 1 shows that potara is unusable after about 100 traces, and that our summarizer increases sublinearly with the number of documents for a particular task.

Macro-scalability While it is important to know how a summarizer works at a document level, we are summarizing entire sets of traces. Therefore we ran scalability experiments where an increasing number of traces were summarized. Since traces can have multiple tasks, this means that each summarization job in this experiment is actually running multiple times, once for each task, before it is considered complete. We ran these experiments on sets with varying number of traces. Table 2 shows the results for the benchmark. Similar to the micro-scalability benchmarks, it is important to note the lack of scalability in potara.

Summary Quality We determined our summaries should be able to capture a general idea of the "regular" execution state of the task being summarized. A task may have hundreds of sentences in its documents, but only 20 that are unique. As a metric for summary quality we checked how many of the unique sentences in a task's corpus were present in a summary. Table 3 shows that Potara is not able to capture the general idea of a regular execution because at max it is using less than 50% of the unique sentences in a tasks corpus of sentences.

5.4 Trace Comparison

Quantiative Analysis We first measured the amount of time taken to generate the diff between the text representations of two traces. From the DeathStarBench, we randomly chose 100 different pairs of traces and measured the time taken to generate the diff between the trace. On average, it took 4.87 milliseconds to generate the diff and measure the distance between two pairs of traces.

Accuracy of Distance function We wanted to ensure that the distance function we had come up behaves in the expected way when computing the distance between pairs of traces. Table 4 shows our detailed results. First, the distance function returns 0 when computing the distance between two identical traces. Furthermore, the distance function grows monotonically with increase in deviation between traces.

Qualitative Analysis of Diff Regarding the quality of the comparison generated, the user men-

[3]Tool available at `https://github.com/sildar/potara`

Number of documents	Num Matched Sentences - Potara	Num Matched Sentences - OS	Total Sentences
5	4	16	16
10	5	19	19
25	7	19	19
100	7	20	20
500	8	20	20

Table 3: Quality comparison of the summary generated by Potara and Our Summarizer (OS) for varying number of documents for a given task

Trace Pair	Distance
Identical Traces	0.0
Non-Error traces of same type (API)	258.0
1 Error, 1 Non-Error trace of same type (API)	1610.0
Traces of different type (API)	1815.0

Table 4: Measured Distance based on our distance function for randomly chosen pairs of traces.

tioned the following:"This is really cool, I'm actually surprised how amenable the complex trace data is to being represented as text. I've always wondered how to visually compare two traces;I really like how you've leveraged some of the more established visual indicators of text comparison (the green+ / red- idiom). With trace diff, I understand now why having a more verbose trace representation in text is useful. It's not interesting by itself, but the diffs provide context for honing in on specific parts of the text." This suggests that the trace diff that we generated using text diff is successful. However, we do believe that it has certain limitations that we discuss in Section 6.

6 Discussion & Future Work

6.1 Lessons Learned

Initially for the text generation task, we wanted to apply textual entailment techniques when generating the text but we realized that this will result in loss of detailed information that is useful for performing comparisons.

For the text summarization task, our plan was to use an out-of-the-box summarizer like BERT or another state-of-the-art multi-document summarizer. But we realized quickly that an out-of-the-box summarizer would not work well with our data as the order of the sentences would not be preserved as well as the fact that the summarizer won't scale to a large number of traces.

6.2 Limitations

Our current text representation has 2 issues. The first issue is that even though the text representation captures the temporal and causal ordering, the text representation fails to capture how much time was spent between two pairs of events. This information is useful for diagnosing performance issues especially latency bugs. The second issue is that the text representation of a task merely presents the information about what happened and does not provide any sort of high level inference about the events that happened. The quality of the text representation is useful for comparison tasks but not useful for humans as a standalone piece of information.

The summary text generated for each task flattens the graph structure by performing a topological sort. This results in there being no indication of branching in the generated text. This is fine for performing comparisons but according to our user study, this information is hard to digest as standalone information.

6.3 Future Work

As part of our future work, we would enhance the text representation of a trace to address the limitations. First, we would introduce a special character that represents a pre-defined time unit. Then, each pair of sentences would be separated by the number of time unit characters that represents the amount of time elapsed between the events corresponding to the sentences.

Moreover, it is sufficiently clear, that the text representation of a trace generated for comparison is only useful for comparison but not necessarily for understanding the execution of a trace. Thus, we would need to develop some inference algorithm that can generate high-level inference over the text representation of the trace to provide the user with

high-level information about the trace.

We also want to enhance our summarization algorithm to somehow include the weights of the edges in the summary graph in the generated summary for each task. This would also lead to us creating a weighted distance function whilst performing comparisons using the aggregated summary of traces.

7 Related Work

Building trace aggregation and trace comparison tools have garnered a lot of effort from both industry and academia in the past decade. All of these efforts have been focused at developing visualization techniques. Pintrace (Karumuri, 2017), the distributed tracing system at Pinterest, uses aggregate analysis to compare two different groups of traces to narrow down the root cause of an error. However, the granularity of the comparison is coarse as the comparison is only performed on the distribution of latencies of the traces in each group. This is only useful for identifying high-level trends. Early work on trace comparison similarly compared latency distributions, but required traces to be structurally isomorphic (Sambasivan et al., 2013); this is rare in practice as most traces are structurally unique (Las-Casas et al., 2019). XTrace (Fonseca et al.) has a trace comparison tool that uses a graph difference algorithm to visualize the diff between the event graphs of two traces. Although, the visualization can highlight unique events in each trace, it fails to explain to the user what these events are. Recent work at Uber (Flanders and Shkuro, 2019) compares the structure of incoming traces with a "good" set of traces to find "bad" traces and then visualizes their difference as a directed graph. This involves extracting features from the "good" traces and then comparing that against a potentially "bad" trace. It is not particularly clear how effective such a visualization is in practice. The techniques we have discussed use textual representation instead of visual representation of traces which we believe is superior for explaining the differences between traces as well as similarities between traces.

Traditional text generation systems generate text from data using hand-crafted specifications and rules for generating text from data (Dale et al., 2003; Reiter et al., 2005; Portet et al., 2009; Turner et al., 2009). However, there have also been a rise in data-driven and neural approaches that aim to learn the working of the text generation modules - content planning, sentence planning, and surface realization - either individually or in a combined fashion (Barzilay and Lapata, 2005, 2006; Konstas and Lapata, 2013; Lebret et al., 2016). Our text generation approach is similar to the traditional approach where we write explicit templates and rules for generating text from a trace. We leave the use of neural and data-driven approaches for text generation from traces as future work.

Text similarity is a widely-studied task with many different techniques and measures (Gomaa et al., 2013). Similarity measures are broadly classified into two categories - string-based similarity and corpus-based similarity. In this work, we choose Levenshtein distance as our similarity measure for trace text comparison as Levenshtein distance accounts for insertions, deletions, and substitutions which corresponds one-to-one with missing events, new events, and reordered events. It additionally has the benefit of accounting for the ordering of the letters/words in the strings which is important for our trace text as the ordering of sentences in the strings for a task inside a trace's text encodes temporal relationship between the events.

Multi-Document Summarization techniques are either extractive or abstractive. Extractive techniques (Xiao and Caren[illegible] 2019; Liu and Lapata, 2019; Erkan and Radev, 2004) extract key sentences and phrases from documents based on some sort of similarity score between pairs of phrases or sentences and uses a ranking algorithm as to select the phrases to be included in the summary. However, extractive techniques do not respect the ordering of the sentences which are temporally relevant for the trace text we a[illegible]ummarizing. Abstractive techniques (Devlin et [illegible], 2018; Bing et al., 2015) formulate facts from the documents and then attempts to rewrite the summary from scratch. These approaches do not fit our end-goal of using the aggregate trace text summary for comparison with other traces. However, we do note that these techniques can be useful for generating a summary that provides a better explanation of the traces to users.

8 Conclusion

In this paper we presented novel techniques for generating an aggregate representation of a set of traces, comparing two traces, and comparing a trace with an aggregate set of traces. Our techniques are based on the insight that the users find text to be easier to understand than any other means

of delivering complex information. To this extent, we present a text generation pipeline that generates a text representation for a given trace. These representations are then used to generate a summary of traces using state-of-the-art Multi-Document summarization techniques. Comparison between two traces, and a trace and an aggregate of traces are modeled as text comparison tasks where their difference is measured as a function of the edit distance between their corresponding text representations.

References

Vaastav Anand and Jonathan Mace. 2019. X-Trace trace dataset for DeathStarBench. Retrieved October 2019 from https://gitlab.mpi-sws.org/cld/trace-datasets/deathstarbench_traces/tree/master/socialNetwork.

Regina Barzilay and Mirella Lapata. 2005. Collective content selection for concept-to-text generation. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 331–338. Association for Computational Linguistics.

Regina Barzilay and Mirella Lapata. 2006. Aggregation via set partitioning for natural language generation. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 359–366. Association for Computational Linguistics.

Lidong Bing, Piji Li, Yi Liao, Wai Lam, Weiwei Guo, and Rebecca J Passonneau. 2015. Abstractive multi-document summarization via phrase selection and merging. *arXiv preprint arXiv:1506.01597*.

Netflix Technology Blog. 2018. Lessons from building observability tools at netflix. Retrieved March 2020 from https://netflixtechblog.com/lessons-from-building-observability-tools-at-netflix-7cfafed6ab17.

Robert Dale, Sabine Geldof, and Jean-Philippe Prost. 2003. Coral: Using natural language generation for navigational assistance. In *Proceedings of the 26th Australasian computer science conference-Volume 16*, pages 35–44. Australian Computer Society, Inc.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Günes Erkan and Dragomir R Radev. 2004. Lexrank: Graph-based lexical centrality as salience in text summarization. *Journal of artificial intelligence research*, 22:457–479.

Steve Flanders. 2019. Opencensus and opentracing are now opentelemetry. Retrieved March 2020 from https://omnition.io/blog/opencensus-and-opentracing-are-now-opentelemetry/.

Steve Flanders and Yuri Shkuro. 2019. A picture is worth a 1,000 traces. Observability Practitioners Summit,KubeCon NA 2019 https://www.shkuro.com/talks/2019-11-18-a-picture-is-worth-a-thousand-traces/.

Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*.

Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.

Wael H Gomaa, Aly A Fahmy, et al. 2013. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18.

Jaeger. Jaeger: Open Source, End-to-End Distributed Tracing. Retrieved June 2019 from https://www.jaegertracing.io/.

Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An End-to-End Performance Tracing And Analysis System. In *26th ACM Symposium on Operating Systems Principles (SOSP '17)*.

Suman Karumuri. 2017. Distributed tracing at Pinteres with new open source tools. Retrieved March 2020 from https://medium.com/pinterest-engineering/analyzing-distributed-trace-data-6aae58919949.

Ioannis Konstas and Mirella Lapata. 2013. A global model for concept-to-text generation. *Journal of Artificial Intelligence Research*, 48:305–346.

Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324.

Rémi Lebret, David Grangier, and Michael Auli. 2016. Neural text generation from structured data with application to the biography domain. *arXiv preprint arXiv:1603.07771*.

LightStep. LightStep: Simple Observability for Deep Systems. Retrieved March 2020 from https://lightstep.com/.

Yang Liu and Mirella Lapata. 2019. Hierarchical transformers for multi-document summarization. *arXiv preprint arXiv:1905.13164*.

Jonathan Mace and Rodrigo Fonseca. Universal Context Propagation for Distributed System Instrumentation. In *13th ACM European Conference on Computer Systems (EuroSys '18)*.

Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*.

Rodrigo Fonseca Jonathan Mace. We are losing track: a case for causal metadata in distributed systems.

François Portet, Ehud Reiter, Albert Gatt, Jim Hunter, Somayajulu Sripada, Yvonne Freer, and Cindy Sykes. 2009. Automatic generation of textual summaries from neonatal intensive care data. *Artificial Intelligence*, 173(7-8):789–816.

Ehud Reiter, Somayajulu Sripada, Jim Hunter, Jin Yu, and Ian Davy. 2005. Choosing words in computer-generated weather forecasts. *Artificial Intelligence*, 167(1-2):137–169.

Raja R Sambasivan, Ilari Shafer, Michelle L Mazurek, and Gregory R Ganger. 2013. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE transactions on visualization and computer graphics*, 19(12):2466–2475.

Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*.

Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report, Google.

Ross Turner, Somayajulu Sripada, and Ehud Reiter. 2009. Generating approximate geographic descriptions. In *Empirical methods in natural language generation*, pages 121–140. Springer.

Twitter. Zipkin. Retrieved July 2017 from http://zipkin.io/.

Wen Xiao and Giuseppe Carenini. 2019. Extractive summarization of long documents by combining global and local context. *arXiv preprint arXiv:1909.08089*.

Availability

The source code is available at https://github.com/vaastav/Tracey and the dataset we used is available at https://gitlab.mpi-sws.org/cld/trace-datasets/deathstarbench_traces.