TraViz: Visualization of Traces in Distributed Systems

Vaastav Anand
vaastav.anand05@gmail.com

Matheus Stolet
stolet@cs.ubc.ca

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. Distributed Tracing is one such cross-cutting technique that correlates events across the system to a specific request by propagating a unique context per system with the 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.

In this project, we propose to create a visualization tool that uses data from traces to succinctly represent the structure and performance of a distributed system. Our tool will allow users to compare the path taken by a group of requests to the path taken by other requests. We believe that a visualization tool that can represent the structure of a group of traces, while also visually encoding relevant performance information, will make it easier for the developers to triage root causes of performance bugs.

2 Data

We have 2 datasets that we will be visualizing using TraViz. The first dataset is called HDFS and contains the **collection of traces** and **collection of processes** from a Hadoop file system used for distributed storage and big data processing and the **corresponding source code**. The second dataset is called socialNetwork and was obtained from the DeathStarBench open-source benchmark for cloud microservices [4]. It contains the **collection of traces**, **collection of processes** as well as the corresponding **source code**. Each distributed system comprises of multiple services that run on machines as different processes. The HDFS dataset has a total of 71,001 traces and 18 processes. The socialNetwork dataset has a total of 22,286 traces and 20 processes.

We discuss the attributes of each **trace**, **process**, and **source code** in detail below.

2.1 Process

The process entity represents a service running in the distributed system. The attributes for Process are shown in Table 2.1. In addition to the attributes listed, each process also comprises of multiple threads which corresponds to an OS or Language Runtime level thread.

2.1.1 Thread

Each thread has a categorical attribute **tid**, which is a unique identifier for a thread within a process.

2.2 Trace

A trace represents a request from a client to a service and shows the path of the request through the distributed system. A trace has five attributes: *id*, *duration*, *start_stamp*, *list_of_tags*, and *list_of_events*.

- *id* of the trace is a categorical attribute used to identify a trace.
- *duration* of the trace is the total time taken to service a request.

Attribute	Type	Description	HDFS Range/Cardinality	socialNetwork Range/Cardinality
id	Categorical	Unique identifier of a process	18	20
name	Categorical	Name of the service the process is running	4	13
host	Categorical	Name of the machine on which the process is running	9	13

Table 1: Attributes of a Process

- *start_stamp* is the unix timestamp of the time when the trace was started.

- *list_of_tags* is a list of human defined keywords that serve as the metadata for the trace. There are on average two tags per trace in both datasets, but the socialNetwork dataset has a total of 8 different tags and the HDFS dataset has a total of 22285 different tags.

- *list_of_events* attribute is a list of the events that happened in a trace. The events in the list are partially ordered [5], with events that caused another event preceding the caused event in the list. Such a partial causal relationship between the events forms a DAG.

For the socialNetwork dataset, the number of events per trace range from 3 to 398 with a mean of 99 events. For the HDFS dataset, the number of events per trace range from 0 to 9697 with a mean of 1427 events.

2.2.1 Event

Events are important things that happen in a system, such as acquiring a lock, sending a request to another server, or performing an update. These events are defined by the developer as instrumentation in the source code. Events can be considered as anything a developer thinks is useful enough to log. The attributes of an event are shown in Table 2.

3 Task

We have identified a total of 4 tasks that we want to support with our viz tool. 2 of these tasks are the main tasks for the project whereas the other 2 are stretch tasks which we will only implement if we have already completed the implementation of the main tasks.

The first main task for this project is comparison. Namely, we want to compare the path and performance of different requests. Each one of our traces represents a request and the collection of events in one trace forms a directed acyclic graph. Our comparison tasks are meant to compare the structure of the DAGs created by the events and the duration of different requests. We want to support three different comparison tasks: one trace against one trace, one trace against many traces, and many traces against many traces. In more abstract terms, we will compare the DAGs formed by the events in different traces by partitioning them into side-by-side views or by showing some sort of a graph diff.

The second major task we are proposing is summarizing data. Many of the traces in our datasets are similar, so we want to aggregate traces with the same tags and events. We believe aggregating traces with the same tags or events will give the user a more generalized understanding of the traces. The user will be able to analyze the average duration of a group of traces, instead of relying on the data from one single trace. This task will take the DAGs formed by the events of different traces and will summarize them by aggregating traces with the same tags or events, so that we can better understand the topology and paths of these graphs.

There are two more tasks we want to support, but may be out of scope for the project so we are leaving these tasks as stretch goals. The first task is creating a dependency graph using the processes in a trace and the second task is adding source code integration to our tool. The *process_name* attribute in our dataset gives the name of a service in a microservice. Developers building distributed systems are interested in understanding the structure of the microservice architecture they are building. We want to consume the list of events in a trace and use the *process_name* attribute in an event to build a graph that links processes that trigger other processes. This information will be presented to developers so that they can discover the dependencies that build their microservices. The second task consists of using the *src_line* and *file_path* attributes to locate the line in the source code that triggered an event. We will do this by providing a hyperlink to the file and line in the github repository.

Attribute	Type	Description
id	Categorical	Unique identifier of an event
trace_id	Categorical	Unique identifier of the trace to which the event belongs
process_id	Categorical	Unique identifier of the process on which the event occured
thread_id	Categorical	Identifier of the thread in a process on which the event occured
hrt	Quantitative	High Resolution (ns) Unix Timestamp
label	Free-Form Text	Developer-added annotation for the event
full_path	Categorical	Full path of the file in the source code where the event was logged.
source_line	Categorical	Line in the source code where the event was logged. Takes the format of Full_path:Line_number.

Table 2: Attributes of an event

4 Scenario

- **Scenario 1**: A developer wants to find out why two similar requests have very different completion times. The developer will select the two traces corresponding to the requests. After the traces are selected, our tool will use the events of these traces to generate a view showing the differences between the traces as well as showing the context and timing of the original requests. This will allow the developer to analyze and identify why 2 similar requests have different completion times.

- **Scenario 2**: A developer wants to find out why requests on Monday are slower than the requests on Tuesday. The developer will make 2 different selections - selection of traces from Monday and selection of traces from Tuesday. Our tool will aggregate the 2 selections into representative graphs and then show the difference of these two aggregate traces. This will allow the developer to possibly figure out a high-level change between the request execution from Monday to Tuesday.

- **Scenario 3**: A developer wants to analyze why a given trace is anomalous as compared to some of the previous traces. To do this, the developer will first create a selection of traces that will be aggregated down into 1 trace. The developer will then select the anomalous trace and create a comparison between the anomalous and the representative trace. Our tool will show the difference between the 2 traces and allow the developer to figure out why a particular trace is anomalous.

- **Scenario 4**: A developer wants to understand the communication load between different services of the system. Our tool will show the developer an overview graph that shows how often 2 services in the distributed system communicate with each other. This will allow the developer to figure out how an addition of a new service would increase the load on each service.

5 Proposed Solution

The events of a trace will be transformed into a node-link graph. One possible idiom for trace comparison, is to compare the graphs formed by the events of different traces by partitioning them into side-by-side views. Another possible idiom is showing the diff of the 2 graphs whilst still preserving the original graphs to provide some sort of context. Because of the limited screen space of a computer monitor, we will allow at most two graphs to be compared at one time. Even though our tool will only allow two graphs to be compared at once, users will still be able to compare multiple traces by aggregating traces that go through the exact same nodes into one graph. In other words, traces that have the same structure can be aggregated into one graph. This aggregation procedure will allow us to make comparisons that involve many traces while only using two DAGs. The links of the graph will be saturated according to the average time it takes for the event in the end of the link to complete. We will also use colour hue to identify the *process_name* of each event. A preliminary sketch of this visualization can be seen in Figure 1.

We will use a similar approach to create the service dependency graph. To create the dependency graph for one trace, we will transform the list of events into a DAG represented by a node-link graph. All the events with the same *process_name* will be aggregated into the same node. In this representation,

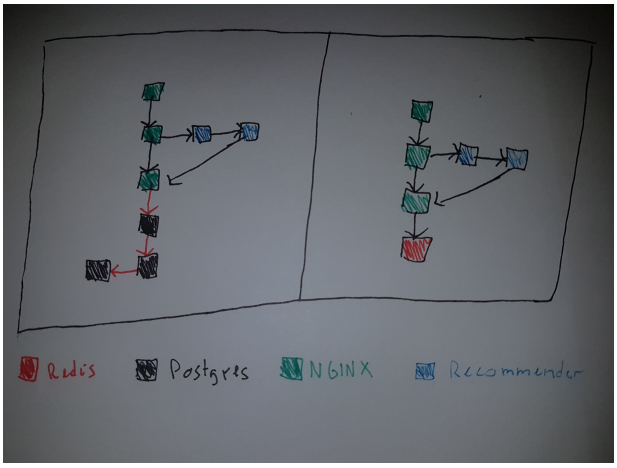

Figure 1: Example of a comparison task where we compare one task that calls Postgres and another that calls Redis. The trace that called Postgres triggers slow events (represented by the red links) and triggers more events in general.

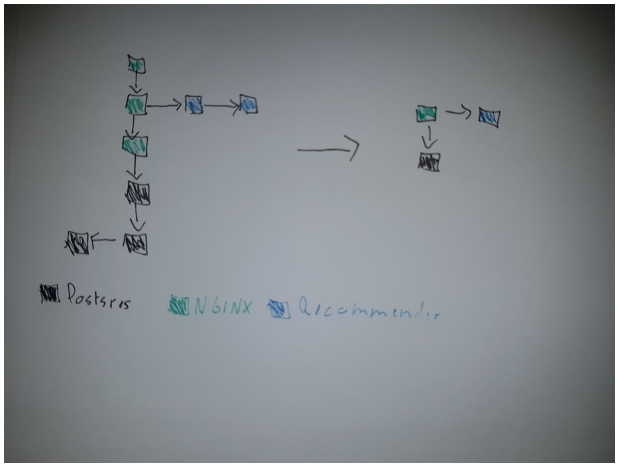

Figure 2: Example of an aggregation task where events with the same *process_name* get aggregated into the same node.

the aggreated node will represent a service in a microservice. The links to the node will be saturated according to the amount of time it takes for all the events in a service to complete. A preliminary sketch of this visualization can be seen in Figure 2.

At the moment, our proposed solution for source code integration does not have a specific visualization in mind. We are thinking about providing a link to the file and line in the github repo corresponding to the selected event in a trace. At least initially, we are not going to do any complicated visualization to accomplish the source code integration task.

6 Implementation Approach

Our visualization tool will be a webapp. We will host a backend written in Go that will serve requests from a client. The frontend will be written in javascript. We will use D3 to manipulate the data served by our backend and to create the visualizations necessary to fulfill our comparison tasks.

7 Schedule

We are prepared to spend 100 hours in total, 85 of which we have specified in Table 3. We are keeping 15 hours as buffer time for tasks that might require extra time.

8 Expertise

Both Matheus and Vaastav have experience with the problem domain. Vaastav has been working on distributed systems for years and has participated in many different projects in the area. Furthermore, his recent research has investigated tracing in distributed systems. Matheus also has experience with distributed systems through coursework, research assistanships and graduate research. Although, tracing is a new topic for him, he has found it to be an interesting area that can be useful for debugging and comprehending the interactions of large networks of computers.

9 Related Work

Jaeger [1] is an open-source distributed tracing project that provides libraries for instrumenting distributed systems in different languages as well as a frontend for viewing the traces produced from these systems. Jaeger has vis idioms for visualizing a single trace, for visualizing the dependencies between services of the system, as well as an idiom for comparing 2 different traces. However, it doesn't have any idiom for viewing an aggregate form of multiple traces or for comparing groups of traces.

LightStep [2] is a start-up company that creates solutions for real-time tracking of requests and metrics in large-scale distributed systems. LightStep has vis idioms for visualizing the structure of a single trace,

Task	Est Time	Deadline	Description
Project Review 1	2	Mon, Nov 18	Prepare slides
Project Review 2	2	Mon, Dec 2	Prepare slides
Implementation	55	Fri, Dec 16	Completed viz tool
- Go backend	4	Fri, Nov 8	Implement a backend server that responds to requests about dataset
- Project frontend	6	Tue, Nov 11	Set up the frontend including dataset selection and selecting groups of traces (allows selection and filtering of traces based on their attributes)
- Main Viz (Comparison)	25	Mon, Dec 2	Implement and design viz for capturing 2 DAGs while preserving content
- Main Viz (Aggregation)	10	Thu, Nov 20	Implement and design viz for aggregating traces
- Stretch Viz (Dependency graph)	5	Fri, Dec 6	Design tool to visualize the communication dependency graph between services
- Stretch Viz (Src Code Integration)	5	Fri, Dec 6	Link each event in trace to line and source code
Presentation	10	Mon, Dec 9	Prepare slides + video (demo)
Final Paper	16	Fri, Dec 13	Finalize paper

Table 3: Project milestones

for visualizing the critical path of a single trace, and for visualizing various metrics collected for multiple traces. Like Jaeger, LightStep also lacks a viz idiom for viewing aggregate traces and for comparing groups of traces.

ShiViz [3] is an interactive visualization tool that visualizes communication graphs from distributed system execution logs. As ShiViz is designed for visualizing logs and not individual traces, ShiViz does not support comparisons of multiple traces.

With TraViz, we want to address some of the shortcomings of existing tools by providing a way for the users to compare the structure of a group of traces as well as integrate the traces with source code to provide more context for debugging.

References

[1] Jaeger - distributed tracing system. `https://www.jaegertracing.io/`. Accessed: 2019-11-04.

[2] Lightstep. `https://lightstep.com/`. Accessed: 2019-11-04.

[3] Shiviz. `https://bestchai.bitbucket.io/shiviz/`. Accessed: 2019-11-04.

[4] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18. ACM, 2019.

[5] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.