TraceZip: A Compression Technique for Distributed Tracing

Muhammad Haseeb
New York University

ABSTRACT

Due to the growing complexity of web services, application developers are switching towards an architecture that decomposes large services into smaller components that are developed and deployed separately. These smaller components, referred to as microservices, use Procedure Calls (RPC) for communication and trace is the term for the history of these RPCs. These traces are used to debug microservice applications, and the practice of collecting the traces is called distributed tracing. Due to the huge volume of the traces, storing them is impractical in terms of storage space and network cost. So the volume of traces that is stored is reduced. One widely used method for achieving this is called sampling. However, reducing the data stored leads to information loss which in turn affects the debugging performance [2].

We propose an alternative system, TraceZip, that allows architects to store all of the traces via efficient compression so that the storage costs stay at a reasonable level. TraceZip is designed around the structural characteristics of traces and the evaluations show that it can help to save around 75% of the storage capacity compared to storing the uncompressed traces.

1 INTRODUCTION

Increasing application complexity has caused applications to be refactored into smaller components known as microservices that communicate with each other using RPCs. Distributed tracing has emerged as an important debugging tool for such microservice-based applications. Distributed tracing follows the journey of a user request from its starting point at the application's front-end, through RPC calls made by the front-end to different microservices recursively, all the way until a response is constructed and sent back to the user. To reduce storage costs, distributed tracing systems sample traces before collecting them for subsequent querying, affecting the accuracy of queries on the collected traces [2].

Almost all of the related literature has focused on developing a smart sampling technique so that only a subset of the traces need to be stored. These sampling techniques can be categorized into two general classes: (i) head-based sampling and (ii) tail-based sampling. In head-based sampling, the decision for storing or not storing the trace is taken before the trace has been completed. In tail-based sampling, the decision is taken after a complete trace is available. However

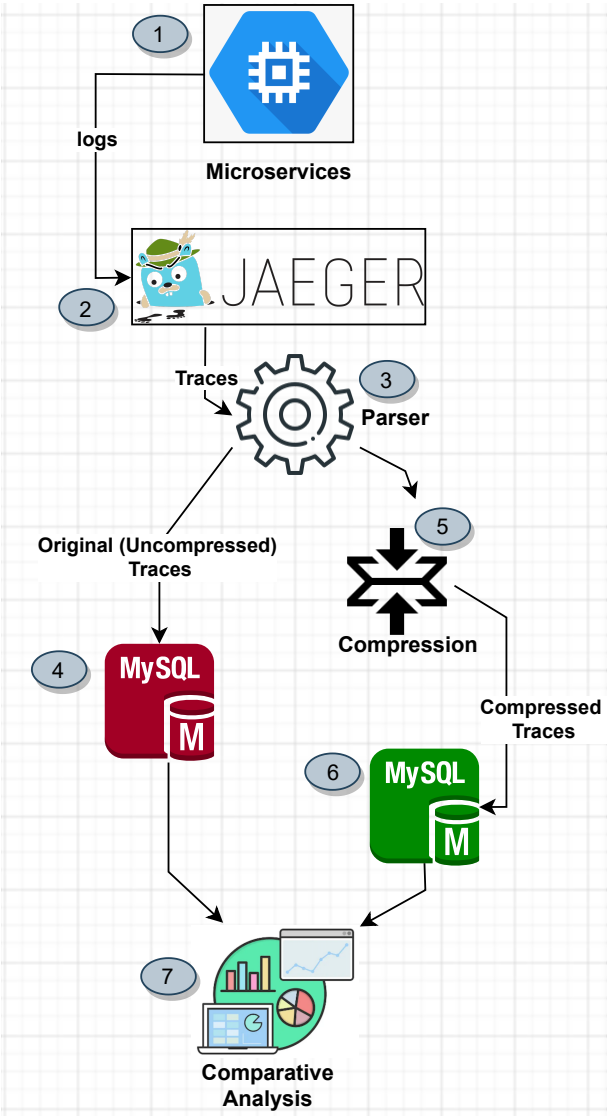

Figure 1: Evaluation Setup

Distributed Systems, Fall 2021, NYU

all the sampling techniques compromise on debugging performance by not storing some subset of the traces. TraceZip works around this compromise by efficiently compressing the traces so that almost all of the traces can be stored, and yet it manages to maintain reasonable storage costs. In fact, the evaluation shows that TraceZip saves around 75% of the storage capacity compared to storing uncompressed traces.

2 RELATED WORK

Almost all of the related literature has focused on developing a smart sampling technique so that only a subset of the traces need to be stored. These sampling techniques can be categorized into two general classes: (i) head-based sampling and (ii) tail-based sampling. In head-based sampling, the decision for storing or not storing the trace is taken before the trace has been completed. In tail-based sampling, the decision is taken after a complete trace is available. Canopy, Dapper, and Jaeger [1, 3, 5] are examples of the former. Lightstep [6] and some parts of Canopy employ different flavors of the latter. Another approach is query-based distributed tracing. This includes Pivot Tracing [4] and Snicket [2]. They do not store the traces as they run the queries in real-time in the microservices and only store the results of the pre-defined queries. The main disadvantage is that they limit the historical visibility into the microservices state as both of these works execute pre-defined queries so only the required data is collected. TraceZip can also be used as an extension of these works for improving the state visibility while maintaining the storage costs at a reasonable level.

Head-based sampling, tail based sampling, Pivot tracing and Snicket, all of them lose because they limit the historical state visibility in one way or another. TraceZip's advantage is that no compromise is made on the historical state visibility.

3 DESIGN

TraceZip uses the structural characteristics of traces to efficiently compress them. Traces are often represented through directed acyclic graphs rooted at the front-end. In the graph representation of traces, individual microservices are represented as nodes and all the communication between the microservices is represented as edges between the nodes. The design was motivated by the insight that these graphs have a lot of common sub-graphs that are repeated within a single trace as well as across several traces. TraceZip identifies common sub-graphs and make identifiers for them. The generated identifiers then replace the corresponding common sub-graphs in all the traces when storing them. Figure 2 shows two traces (Trace 1 and Trace 2) where each have two sub-graphs that are common in both the traces. Figure 3 shows that the Trace 2 have pointers to the already stored sub-graphs and need not store the duplicate sub-graphs again. This way TraceZip avoids storing duplicate data that leads to saving substantial storage capacity.

4 EVALUATING TRACEZIP

TraceZip is evaluated on traces generated by microservices consisting of 6 nodes that communicate with each other for serving user requests. We setup a microservices site using a demo application, HotROD provided by Jaeger[1] that is run on a Docker container. Figure 1 shows our setup that is used for evaluating TraceZip. The first component in the setup is the microservices application that generates traces and the second component, Jaeger, consumes the traces and outputs them as graphs. The third component is a parser that strips off some information (e.g., canonical IDs, timestamps) that the current version of TraceZip does not compress. The fourth component is a MySQL database that stores the uncompressed traces' output by the parser. The fifth component, TraceZip does the compression. The sixth component is another MySQL database that stores the compressed traces. The seventh component analyzes the two database. This setup is used for the analysis of TraceZip on 400 traces. Figure 4 shows the comparison between sizes of the two databases, one database containing the uncompressed traces and the other database containing the compressed traces. The compressed traces require substantially lower storage than the uncompressed traces. Figure 5 shows the percentage decrease in the size of traces after the TraceZip compression. As the number of traces increase, the percentage decrease in size also increases.

5 FUTURE WORK

TraceZip has been evaluated on an SQL database and for comparison it has been compared against another SQL database that stores the original traces. The next step is to compare it against standard databases that compress the data while storing it, for example Cassandra, gzip etc,. The time taken for compressing the traces and then decompressing them for answering the queries need to be studied so that constitutes another next step. At the moment, timestamp information is stripped off the traces however we aim to retain it by doing proper compression on it, potentially by using base values and deltas of the subsequent timestamps.

5.1 Limitations

TraceZip currently takes significant time for compression that has not been benchmarked yet. As we intend to study the time taken for compression and decompression, our observations have shown the need to optimize TraceZip so that it scales well against the huge volume of traces.

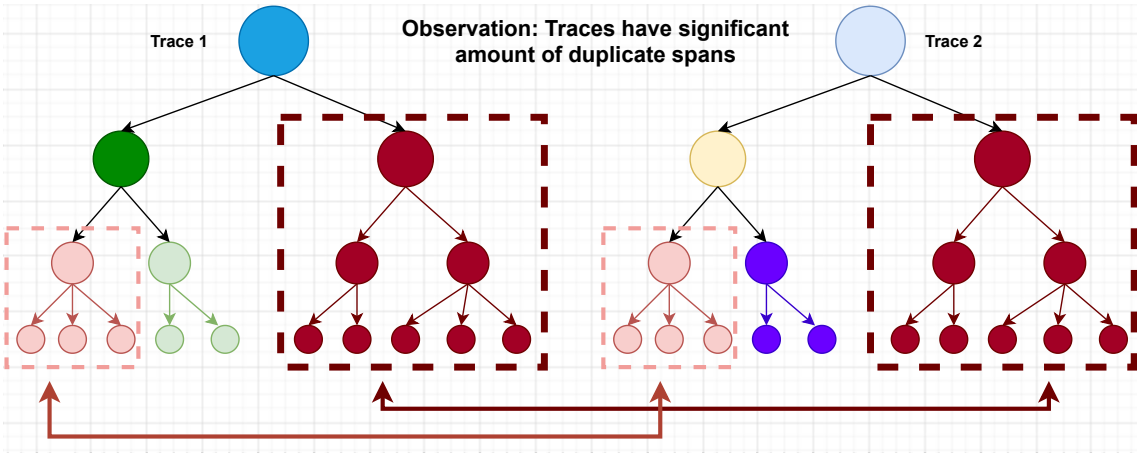

Figure 2: Two traces where each have common sub-graphs; the red sub-graph and the pink sub-graph

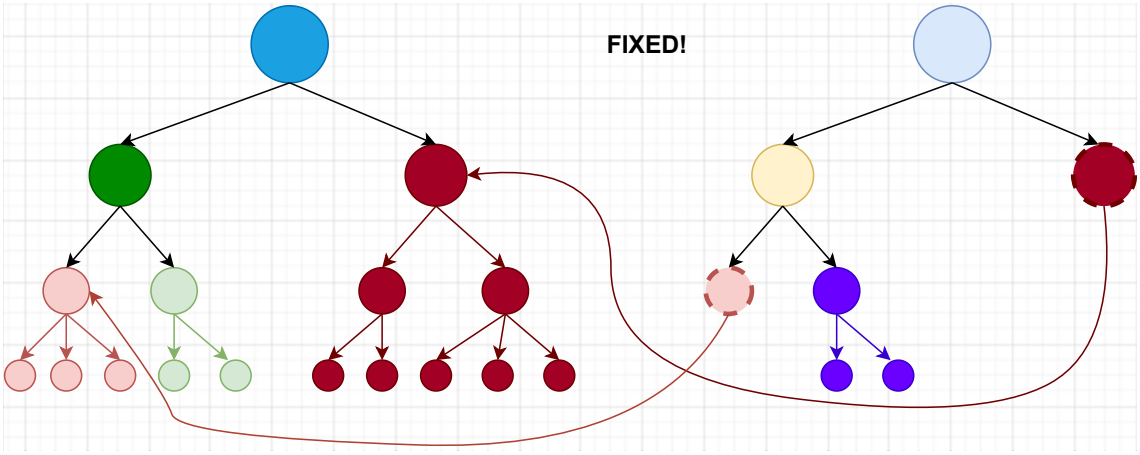

Figure 3: The identified common sub-graphs are not stored more than once, instead the new traces will make a pointer to the already stored sub-graph

6 CONCLUSION

We have presented TraceZip, a compression technique that takes advantage of the structural characteristics of traces and efficiently compresses them. It makes it possible to store all of the traces and still manage to maintain the storage costs at a reasonable level. The evaluation shows that TraceZip can help in saving around 75% of the storage capacity compared to storing the uncompressed traces.

REFERENCES

[1] Cloud Native Computing Foundation. 2021. Open Source, End-to-End Distributed Tracing. https://www.jaegertracing.io/. Accessed: 2021-06-25.

[2] Berg et al. 2021. Snicket: Query-Driven Distributed Tracing.

[3] 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. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *ACM SOSP*.

[4] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *ACM SOSP*.

[5] Benjamin H. Sigelman, Luiz André 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, Inc.

[6] Robin Whitmore. 2021. How Lightstep Works. https://docs.lightstep.com/docs/how-lightstep-works. Accessed: 2021-06-25.

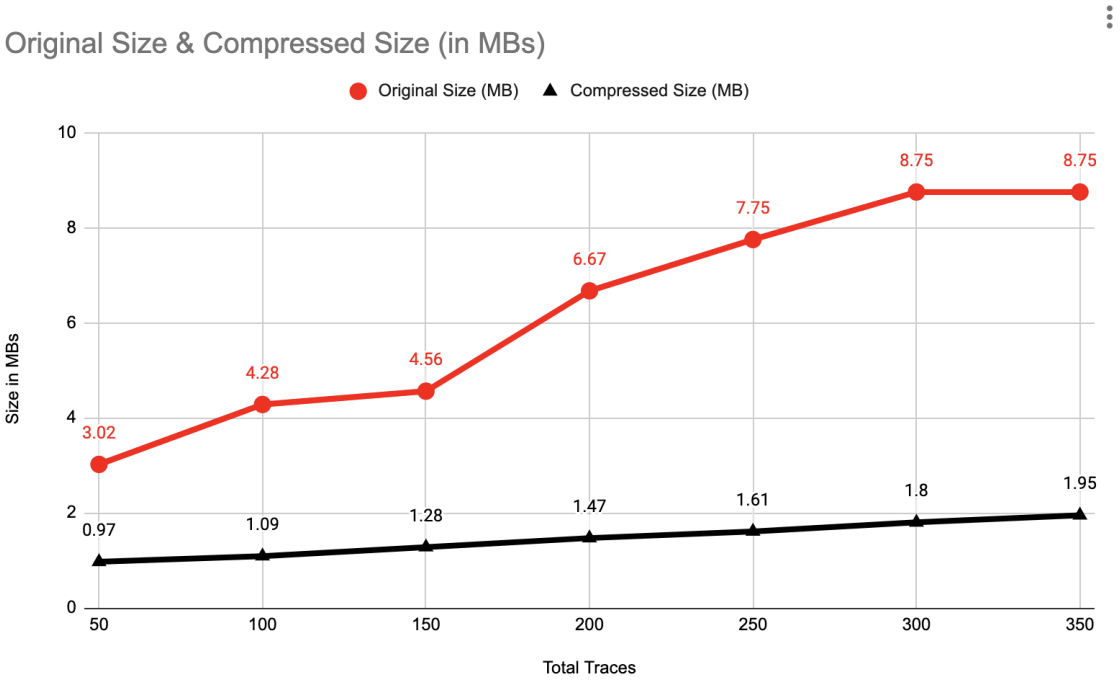

Figure 4: Comparison of uncompressed traces and compressed traces

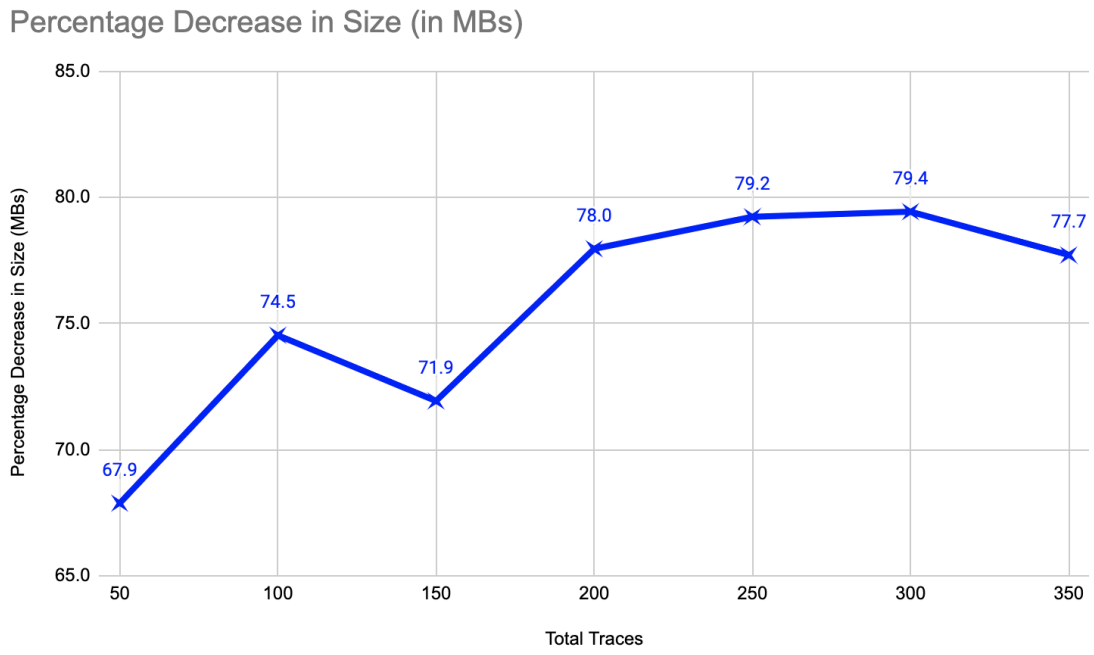

Figure 5: Percentage decrease in the size of traces after TraceZip compression