Locating regions of uncertainty in distributed systems using aggregate trace data

MIT PRIMES 2022 Final Report

Jiayi Dong and Anshul Rastogi

MIT PRIMES 2022 Computer Science Division
Mentors: Darby Huye, Max Liu, Zhaoqi Zhang, Dr. Raja Sambasivan

Abstract -

Distributed systems are central to countless applications in the modern world. These applications can have tens to thousands of components interacting making it difficult to identify the source of performance problems. Distributed tracing is widely used to elucidate the interactions within a distributed system; however, instrumenting system codebases can be tedious, and collecting tracing data generates overhead. Optimally, minimal instrumentation is added to regions of the codebase that explains the majority of the system's performance variation. We present a prototype application that highlights regions of performance uncertainty in a system, guiding developers to where instrumentation would most increase predictability. Using aggregate trace data, spans are ranked by uncertainty metrics, which are primarily the standard deviation and coefficient of variation of the exclusive latencies of an operation across multiple traces. We developed our prototype in Python and applied it to trace data extracted from HotROD. We evaluated our tool on four test scenarios where we injected latency into services in HotROD. Our tool highlights the service(s) with injected latency in all four test cases.

Keywords -

Distributed Tracing; Jaeger; HotROD

1 Introduction

Distributed systems are an integral component of our modern world. They span from broader systems such as cellular networks to specific services such as Google or Facebook. These services often involve various interacting components – indeed, they can easily contain thousands of microservices – so developers that wish to modify, regulate, or alter the distributed system must be aware of the interactions between these components and how altering one may affect others. This is similarly important for addressing errors in the system. To do so, they must be able to reduce their *uncertainty*, or the lack of ability to predict an outcome of an interaction (such as end-to-end latency) with given information.

To decrease uncertainty for a system with so many components, developers must increase the amount of information that they have. In obtaining this information, developers rely on distributed tracing, which gives them insight into the interactions comprising a distributed system by tracking system calls and logs. This facilitates the modification and regulation of the system, which is particularly important in projects where several developers regulate specific applications in a codebase, as is seen in industrial-level projects. No one developer can possibly be aware of the countless interactions, yet these developers may need to make modifications to their specific microservices that could affect other components of the system. Tracing data allows developers to view the communication and system calls amid these services to better approach their specific areas.

Another common issue for distributed systems is diagnosis of issues in system performance. Tracing allows for insight into these systems for various developers, allowing them to understand the interactions within the system and the ramifications of modifications that they may make, such as altering a component in a microservice. Localizing resource bottlenecks, cache misses, and other issues can be determined with the information found in trace data.

However, acquiring tracing data generally requires manual instrumentation of the system. This can be problematic since it may be difficult for developers to determine where instrumentation should be placed to be both meaningful and useful to them for system regulation and/or modification. For instance, they might miss instrumenting an important resource bottleneck that could have informed them on a critical latency issue. Conversely, they may instead instrument areas that are of little use to the developers and oversaturate logs with unimportant information that they will then have to sift through. These issues arise from improper placement of trace points, which is discussed in further detail in 2.4.

Definition 1.0.1 (Trace Points)**.** A *trace point* is an instance of instrumentation at a particular point in a distributed system codebase, such as a log.

Definition 1.0.2 (Spans)**.** “The *span* is the primary building block of a distributed trace, representing an individual unit of work done in a distributed system. Each component of the distributed system contributes a span — a named, timed operation representing a piece of the workflow” [1] whose boundaries are determined by trace points.

In this paper, we present a project that aims to aid developers in the placement of trace points to effectively reduce uncertainty in regards to system latency. We developed an algorithm that processes span-based trace data and ranks spans by their degree of uncertainty they contribute. As such, this points developers to areas that may benefit from greater instrumentation.

Thus far, our Python prototype has been moderately successful at identifying spans with highly variable latencies (caused, for instance, by intermittent errors) in a variety of cases given the trace data produced by requests to a simplified distributed system simulation. Of the four scenarios tested, the protoype identified the spans with the injected uncertainty in all four cases.

2 Background

2.1 Our Tools

As mentioned in the introduction section (see 1), tracing is an essential component to debugging when it comes to distributed systems because microservices are so scattered across the different nodes in the system that it becomes impossible to immediately find the causation of an error, latency, or a memory overflow. With a tracing tool that visualizes spans, we are able to sort requests in a logical hierarchy and see them side-by-side in chronological order, see the timestamp and location of errors, and visualize the logs that are recorded as spans are running. The distributed tracing visualization tool that we primarily use in our research is Jaeger (see Figures 1 and 2).

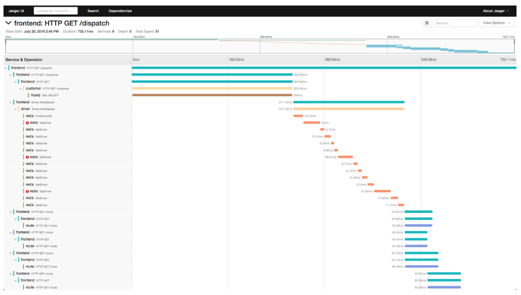


Figure 1. Example of Jaeger’s visualization of spans in a Gantt Chart

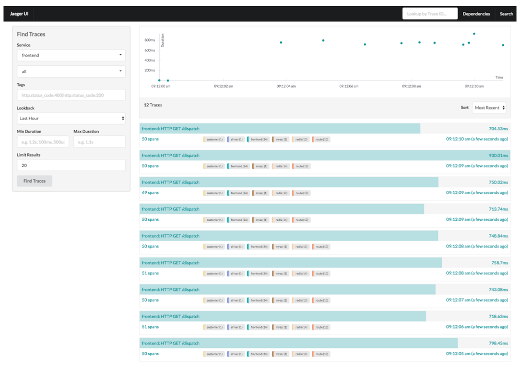


Figure 2. Jaeger’s data visualization for trace data from HotROD

Jaeger, created at Uber [2], is an open-sourced, end-to-end distributed tracing visualization tool that is capable of monitoring distributed-system-based microservices. Being language-independent, Jaeger can track RPCs (remote procedure calls) between microservices in all components of a distributed system, and is used at the industrial level in services, including Uber. Jaeger can visualize spans as well as logs and errors in distributed programs as they run in real-time. Some particularly useful features for this project include root cause analysis, latency optimization, and distributed transaction monitoring. Most importantly, however, Jaeger is also used to extract much of the trace data we use, as the visual representations, while important, can be difficult to digest when generalized to the complexity of the great majority of distributed systems.

We commenced our experiments on application simulations that provided support with Jaeger. The simulation we used was one developed by the Jaeger team called HotROD. HotROD simulates a service similar to Uber: upon user request, the application finds drivers nearby and orders a ride. It then shows the estimated time for the ride to arrive. HotROD is easy to test on, and we also found in its code that the way it simulates errors is very predictable and thus straightforward to work with.

2.2 Previous Work

Numerous applications have been previously developed with the goal to assist developers in processing trace data for instrumentation. Some approaches include selectively storing trace data to keep more relevant or useful information for debugging. Techniques for such software include approaches as unique as machine learning, as employed in Sifter [3] to determine more anomalous or edge-case request paths in trace data, while projects such as VAIF [4] rely on custom data structures seek to enable the logs most relevant for analyzing performance issues. Both Sifter and VAIF decrease the amount of data developers would need to sift through to identify anomalies or performance issues. Isolating relevant data is often crucial as discovering issues can otherwise quickly become a needle-in-a-haystack problem. Exhaustively saturating a codebase with trace points would be tedious, generate incompatible levels of overhead, and the sheer data produced could not be practically or effectively analyzed manually by a human developer. Here, VAIF in particular emphasizes the significance of the placement of trace points and logs in the role of localizing potential issues in the system.

In addition to paring back tracing data, approaches such as graph-based microservice trace analysis (GMTA) [5] improves data recovery, workflow analysis, and user interactions with its graph-based data structure for storing trace information. GMTA efficiently processes incoming trace data on-the-fly in a way that is easily and effectively scalable for industrial-scale microservice-based systems. Relying on a unique data structure could be an important insight for distributed tracing.

One central goal in various projects regarding trace data is critical path analysis (CPA).

Definition 2.2.1 (Critical Path)**.** A *critical path* is the longest path in a directed acyclic graph (DAG). In trace data, this represents a singular path that accounts for the end-to-end latency of a request path.

Critical paths are a key approach to optimizing systems as they define the maximum length to request latencies. Thus, to optimize a request latency, the developer would want to focus on optimizing elements along the critical path. Each trace of the system will contain a critical path, which can be determined algorithmically with relative ease, such as by a recursive function [6]. However, the critical path's structure, length, and other properties may vary widely across separate requests; developers seeking to achieve service-level objectives for request latencies may be unable to identify specific regions of the codebase to optimize. CPA seeks to create a comprehensive, system-wide interpretation of critical paths. Rather than extracting the critical path of one trace, CPA tools such as CRISP [7] and *The Mystery Machine* [8] utilize aggregate trace data from thousands of requests to distill a more complete picture of the system. Moreover, using aggregate data can provide a model for system interactions that individual traces, being request-specific, cannot. *The Mystery Machine*, for instance, can define various relationships between thread segments in a system as defined by their consistency across traces.

In addition to the use of aggregate data, the tool CRISP offers another important insight via the introduction of two metrics: inclusive and exclusive latencies. Inclusive and exclusive latencies are defined as follows for an operation (span):

Definition 2.2.2 (Inclusive Latency)**.** The *inclusive latency* of a span, S, is "the wallclock time elapsed in the S itself, plus the wallclock time elapsed in the transitive closure of all other operations S calls. This is also purely the duration of execution of S." [6]

Definition 2.2.3 (Exclusive Latency)**.** The *exclusive latency* of a span, S, is "the wallclock time spent in S itself." In CRISP, the exclusive latency "neglects the time spent in S's children operation(s), say T, if T appears on the critical path." [6] For our purposes, T's latency is neglected even where it does not appear on the critical path.

An illustration of exclusive latency can be found in Figures 3 and 4. In both figures, the zigzagging arrows represent RPC send and receives. In Figure 3, we see the parent-child relationship between spans S and T. In Figure 4, Span S is segmented by the moments where it sends and receives the RPC to span T; the exclusive latency is the sum of the durations S_1+S_3 whereas S_2 is neglected in this metric.

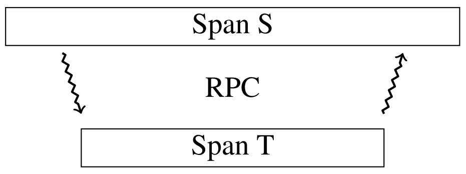


Figure 3. Children-parent relationship between spans A and B.

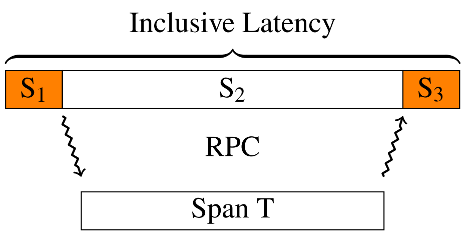


Figure 4. Segment-based interpretation of spans S and T. Here, the exclusive latency of operation S is the sum of the orange intervals.

The purpose of exclusive latency in particular is the extract the amount of wallclock time that the execution of S alone accounts for, thereby localizing latency issues. For instance, we can infer that high uncertainty in the exclusive latency of S originates solely from S. In contrast, high variability in the inclusive latency of S could be attributable to the execution of a child span of S, where S is waiting for a response from its child operation prior to resuming its own execution. This is a valuable insight that is central to our own project.

2.3 Event-Based vs. Span-Based Models

Trace data is conventionally organized through the span-based model of tracing [9]. This is the model seen throughout numerous tracing softwares, such as Zipkin, Jaeger, and Dapper [10, 2, 11]. However, some softwares, such as X-Trace [12], opt for an event-based model.

A span-based model operates on intervals of time, encapsulating function executions. However, there are numerous limitations to this approach. An article on The Medium [13] remarks on this:

"Spans represent time durations, but the span model gives no specific advice on what durations to measure. Mentally, it has its origin in the (synchronous remote) procedure call programming model, in which spans nicely map to function calls: a function or code block is entered (span starts) and then left (span finishes), in the meanwhile we record on the span what happened. As soon as the programming model is a bit different, there is no trivial way to map it to spans.

"What if the remote procedure call is asynchronous? Shall we finish the span when the request is sent and open a new one for the reply? What if, in the meanwhile, the execution context is transferred to another processing unit and the reply is processed there?

"What if the programming model is messaging? Should the span start when the message is produced or when it is dequeued? Should one or two spans represent the sending and receiving of a message? How about one-to-many communication like publish and subscribe?"

```
 // running on machine 1:

// assume doingWhatever has span identifier and trace identifier values
// of 1 and 5 respectively
Span doingWhatever = Trace.startSpan("doing whatever");

// assume 'sendPing' has span identifier and trace identifier values
// of 1 and 2. its parent span field would be 5 because
// the doingWhatever span is its parent.
Span sendPing = Trace.startSpan("sending ping");

// rpcPing would append the information (trace: 2,span: 1)
// along with the RPC information
rpcPing();
// end the spans and deliver them to some source for processing etc.
sendPing.stop();
doingWhatever.stop();


// running on machine 2:
rpcReceivePing(RPC rpc) {
      if (rpc.hasTracingInformation()) {
            // pingSpan's trace id would be 2, its span id would
            // be randomly generated, and its parent id would be 1,
            // signifying that sendPing is its parent because
            // sendPing's span id is 1
            uint64 traceId = rpc.getTraceId(); // 2
            uint64 spanId = rpc.getSpanId(); // 1
            Span pingSpan = Trace.startSpan(traceId=traceId,
                                    parentSpanId=spanId,
                                    description="rcvd ping");
            // do some stuff then stop and deliver the span
            pingSpan.stop();
      }
}
```

Figure 5. Pseudocode example of span model implementation in RPC mechanism. Adapted from [9].

Instead, we can turn to the event-based model. An event is defined as follows:

Definition 2.3.1 (Event)**.** An *event* is a single, ordered piece of data produced by a single trace point.

As per Definitions 1.0.2 and 2.3.1, spans are comprised of three components – an event at the start, an event at the end, and the interval of time between these two events. The last of these three can always be inferred from the other two, so a solely event-based model can always represent everything that a span can. However, the converse is not true; events allow for finer articulation in trace data due to the ability to add annotations and logged events to demarcate algorithm progress, errors, cache misses, and other single-point occurrences of note. As such, a span, which is required to contain a single unbroken interval, is not as expressive as events may be. Additionally, events have a strict ordering, thus allowing interactions to be defined much more rigidly (such as through Lamport's definition of happens-before relationships [14]). Spans, in contrast, may overlap.

Indeed, a comprehensive 2014 analysis [9] rigorously showed that events are more expressive of system interactions in end-to-end tracing in every aspect, although spans do have the advantage of greater mindshare and more simplistic human interpretation. For instance, the visualization software we use for our project (Jaeger) is tailored only for the span model.

The same 2014 analysis suggests the use of a joint model, incorporating spans into instrumentation where possible but using events wherever greater expressiveness is required to convey information. The paper's solution requires a library that supports instrumenting both spans and events; the current library we are using, the Opentracing API for Go, does not support logged events, although we are searching for workarounds.

For our purposes, we use spans to encapsulate operations and full microservice executions. Although internal logged events could provide details, our dependence on simplified distributed systems means that no such logs will be present. This preserves the simplistic, easy-to-identify (for the interpretation of human developers) structure of the distributed system in the trace data while allowing Jaeger to still visualize our tools' trace data in at least some capacity. Additionally, most industry tracing infrastructures only support the span-based model, so it is the ideal route for this project. Nevertheless, an event-based model remains an intriguing avenue.

2.4 Motivation

One crucial aspect of instrumentation is the placement of trace points. If instrumentation neglects proper placement, a developer may not have any trace points at an important bottleneck and instead have trace points in other areas less relevant to system performance, thus missing crucial information. On the contrary, if instead the system is oversaturated with trace points, the developers viewing the resultant trace data may have to sift through a great deal of extraneous data before finding the relevant logs; additionally, instrumentation could take much longer due to the large number of trace points in this case.

Optimal placement for trace points is informed largely by the usefulness of those trace points to a human developer for reaching desired goals for a distributed system. These goals generally include minimizing latency and/or critical path latency, minimizing/identifying errors, and satisfying service-level objectives (SLOs). Thus, to be useful, a trace point must be able to inform a developer accordingly. Ideally, trace points should identify problematic areas such as errors/anomalies, common cache misses, areas responsible for significant amounts of latency, and the like.

In our case, optimal positioning for trace points are those that most decrease uncertainty. Therefore, we must localize areas with the highest uncertainty and advise developers to place trace points there. To do so, we will employ aggregate trace data of the system, similar to CRISP, and using exclusive latency should allow an algorithm to identify the specific span executions accounting for the uncertainty across various traces.

3 Design

3.1 Assumptions

For our design, we assume the following regarding our trace data:

- There are no asynchronous spans.
- No clock skew occurs. This is an appropriate assumption for HotROD as HotROD runs entirely on an a single, local machine. However, this assumption is not generalizable to real distributed systems.
- Spans/operations pause their execution when waiting for responses from their children. This assumption should hold true for single-threaded

services and simplifies the cases needed to determine exclusive latency. Part of the reason for this simplification is that we want our prototype to function on trace data from minimally instrumented systems. For us, this translates to only receiving topological information of the spans in the request paths with no internal logs.

- RPC latency is 0; for a minimally instrumented system, we do not have enough information to determine the RPC latency in any manner, so this assumption seeks to simplify any potential issues here.

3.2 Extracting Uncertainty

To assist a developer in placement, we must be able to determine the span executions accounting for the most uncertainty in a system. We wish to provide these suggested placements based on comprehensive, system-wide data so that the optimal placement is applicable to various request types. As such, we will use aggregate trace data to localize areas of uncertainty, as inspired by CRISP and *The Mystery Machine* (see 2.2) and the broader insights in to the system this allows.

Additionally, localizing uncertainty to a particular span relies on exclusive latency (see 2.2), as this allows us to causally determine the source of latencies. We can only group these latencies across traces by the operation names of the spans as we do not have internal logs for the spans to classify them further.

Ultimately, however, we must direct developers to areas with the highest uncertainties. To perform this, we must develop metrics for uncertainty. We define these as given in Definition 3.2.1.

Definition 3.2.1 (Uncertainty)**.** The *uncertainty* of a span is a statistical measure of spread (variance) of the span's exclusive latency across multiple traces. In this project, we primarily rely upon the standard deviation of the exclusive latencies. However, standard deviations may appear very large but instead be attributable to natural variation of a very large mean exclusive latency. To avoid misperception of the statistic, the coefficient of variation – that is, the standard deviation divided by the mean – is also calculated so that the developer may compare relative uncertainties.

Upon determining the uncertainties of the various spans, we can rank these spans for the user by uncertainty so that developers can determine which spans to prioritize for instrumentation.

Our algorithm for our prototype is listed in Algorithm 1.

Algorithm 1: Outline of pseudocode for determining uncertainty measures for each span across the various traces.

```
$\alpha \rightarrow \beta$=empty map of spans ($\alpha$) to lists of latencies ($\beta$);
for each trace $\tau$ do
    for each span $S \in \tau$ do
        if $S / \in \alpha$ then
            $\alpha \pm S$;
        for each span $T \neq S \in \tau$ do
            if $S$ is $T$'s parent then
                Update $S$ accordingly;
        Calculate exclusive latency $\phi_S$ of $S$;
        Add $\phi_S$ to list in $\beta$ corresponding to $S$ in $\alpha$;
for each list of exclusive latencies $\ell \in \beta$ do
    Calculate uncertainty metrics;
Rank spans in $\alpha$ by corresponding uncertainty;
```

4 Evaluation

4.1 Implementation

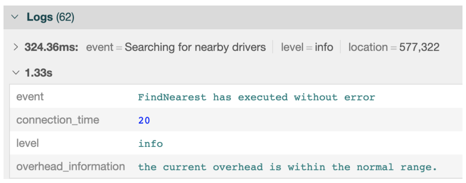
To evaluate our algorithm, we constructed an app implementation. We decided to base our app on Python to expedite development of the prototype. We used the streamlit library for app UI elements and display.

To test our algorithm, we altered the HotROD codebase to create four different scenarios with manually injected latency (§ 4.2). For each scenario, we generated at least thirty requests using HotROD's user interface. We extracted these traces using one of Jaeger's APIs, which downloads all traces that contain a specific span.

The following example code adds a log in a span for the function `FindNearest` when no error has been returned.

```
if err == nil {
    s.logger.For(ctx).Info("FindNearest has executed without error",
                        zap.Int("connection_time",
                                200),
                        zap.String("overhead_information",
                                "the current overhead
                                is within the
                                normal range."))
            }
```

The effect can be seen in Figure 6.



```
Logs (62)
> 324.36ms:  event = Searching for nearby drivers | level = info | location = 577,322
v 1.33s
  event                  FindNearest has executed without error
  connection_time        20
  level                  info
  overhead_information   the current overhead is within the normal range.
```

Figure 6. Effect of code on HotROD trace data as seen in Jaeger

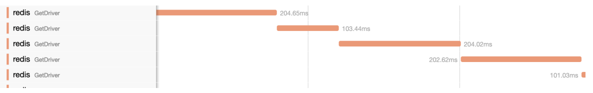


Figure 7. The affected spans of Scenario 1 visualized in Jaeger.

4.2 HotROD Scenario Testing

The latencies that were manually programmed into each scenario are detailed as follows:

Scenario 1: This scenario focuses on manipulating the latency of the operation `GetDriver` under the service `redis`, which the service *driver* often makes a series of calls to. Given how simple it is given it only affects a single operation in a single way, scenario 1 serves as a test of the most basic latency issue that could happen to a system. To first keep out any other potential variables that could impact our result, `redis|GetDriver` (the format of a span's name as presented in Jaeger and our tool is in the format of `service name|operation name`) was set so that its normal latency would be around 100 milliseconds. This was done by editing the values `RedisFindDelay` and `RedisFindStdDev` from HotROD's config file to `100 * time.Millisecond` and `0 * time.Millisecond`, respectively, then applying these values to Golang's native delay.Sleep() function. The manually programmed latency in this scenario was that `redis|GetDriver` had exactly a 50% chance to experience an latency issue that lengthens its latency to approximately 200 ms, while during the other 50% of the time, it works flawlessly and has a latency of 100 ms. This can be seen clearly in Jaeger's visualization, as shown in figure 7. This also means that `redis` inevitably has a bimodal distribution in latency. The purpose of this scenario was to test out how well our algorithm can detect latency issues within each trace itself, which certainly doesn't cover how all latencies function. A manually created latency issue that continuously influences the same operation across multiple traces is presented and tested in Scenario 2.

Scenario 2: Departing from scenario's simpler latency issue whose effects are only seen within a single trace, scenario 2 focuses on a latency issue that increases latency from one trace to the next. In this scenario, the manually injected latency causes each trace themselves to increases by around 100 ms from the last by causing `frontend` to lag by 150 ms. For example, if the first generated trace has a latency of 260 ms, then the next one would be 410 ms, the next 560 ms, as depicted by Jaeger in 8. Therefore, the distribution of latencies in all the traces in this scenario would be relatively equal. `redis` is kept constant at 100 ms to eliminate unnecessary variation. This scenario is different from the last one in that its latency issue increases the latency of a span trace after trace. It tests whether our algorithm is able to differentiate trace-by-trace issues than the kind from Scenario 1, where the issue causing the latencies remains within the trace itself.

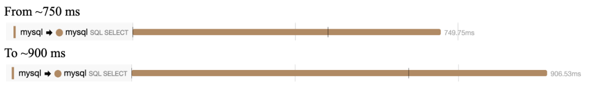


Figure 8. The affected spans of Scenario 2 visualized in Jaeger.

While Scenarios 1 and 2 are able to evaluate the algorithm's performance on detecting one single latency that exists within the tool, it is important for our algorithm to also be able to correctly identify and locate multiple latencies. Therefore, scenario 3 and 4 were created to test out our algorithm's ability in tackling the challenge of two interconnected issues.

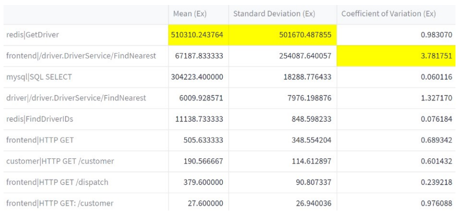
Scenario 3: In this scenario, each trace increases 100 ms from the last (the latency in Scenario 2) and `redis` has 50% chance of being 500 ms, and the other 50% of chance being 1000 ms. Therefore, the two latencies are independent of each other.

Scenario 4: Similar to Scenario 3, each trace increases 100 ms from the last. Different from Scenario 3, however, a conditional was added. If and only if the length of the full trace is smaller than 800 ms, will `redis` have the latency of 100 ms. Else, if the length of the full trace is bigger than 800 ms, an extra latency is added to `redis`, making it 200 ms. Therefore, in this case, the two latencies are dependent on each other in that one only happens if the other also happens.

4.3 Results

Performance on Scenario 1:

As shown in Figure 9, our tool generates a table that displays the ranked uncertainty measures for each span within the trace. The table correctly highlights the span `redis|GetDriver` as the one with the highest uncertainty overall, and therefore the one that the user should prioritize placing trace points in. Since the artificial latency was indeed injected into `redis|GetDriver`, this demonstrates that our algorithm has picked out the correct span to have the developer focus on.



	Mean (Ex)	Standard Deviation (Ex)	Coefficient of Variation (Ex)
redis|GetDriver	510310.243764	501670.487855	0.983070
frontend|/driver.DriverService/FindNearest	67187.833333	254087.640057	3.781751
mysql|SQL SELECT	304223.400000	18288.776433	0.060116
driver|/driver.DriverService/FindNearest	6009.928571	7976.198876	1.327170
redis|FindDriverIDs	11138.733333	848.598233	0.076184
frontend|HTTP GET	505.633333	348.554204	0.689342
customer|HTTP GET /customer	190.566667	114.612897	0.601432
frontend|HTTP GET /dispatch	379.600000	90.807337	0.239218
frontend|HTTP GET: /customer	27.600000	26.940036	0.976088

Figure 9. Uncertainty table of spans for Scenario 1.

As shown in the histogram in Figure 10, the bimodality of the latency in `redis|GetDriver` was also correctly identified, as the histogram shows two clear peaks without outliers. This demonstrates that our algorithm not only has the ability to correctly identify the span with the highest uncertainty but is also capable of describing the specific pattern that these spans have.

Performance on Scenario 2:

As shown in Figure 11, our algorithm identified `mySQL|SQL Select` as the spans with the highest latency. As shown from the information of the injected latency, this is correct, displaying that our algorithm correctly identified the spans that is the most uncertain in the trace. Both latency issues within traces themselves and latency issues from trace to trace has been correctly identified and highlighted.

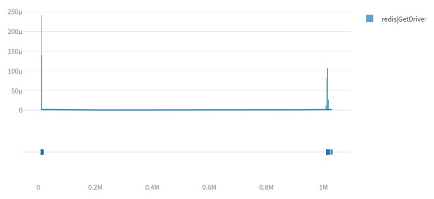
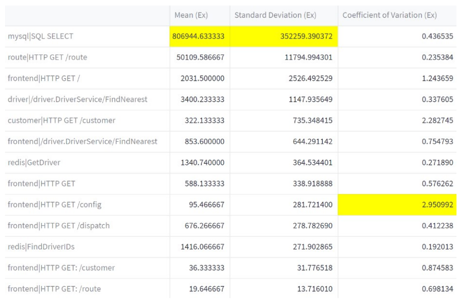


Figure 10. Histogram of latency for `redis|GetDriver` in Scenario 1



	Mean (Ex)	Standard Deviation (Ex)	Coefficient of Variation (Ex)
mysql|SQL SELECT	806944.633333	352259.390372	0.436535
route|HTTP GET /route	50109.586667	11794.994301	0.235384
frontend|HTTP GET /	2031.500000	2526.492529	1.243659
driver|/driver.DriverService/FindNearest	3400.233333	1147.935649	0.337605
customer|HTTP GET /customer	322.133333	735.348415	2.282745
frontend|/driver.DriverService/FindNearest	853.600000	644.291142	0.754793
redis|GetDriver	1340.740000	364.534401	0.271890
frontend|HTTP GET	588.133333	338.918888	0.576262
frontend|HTTP GET /config	95.466667	281.721400	2.950992
frontend|HTTP GET /dispatch	676.266667	278.782690	0.412238
redis|FindDriverIDs	1416.066667	271.902865	0.192013
frontend|HTTP GET: /customer	36.333333	31.776518	0.874583
frontend|HTTP GET: /route	19.646667	13.716010	0.698134

Figure 11. Uncertainty table of spans for Scenario 2.

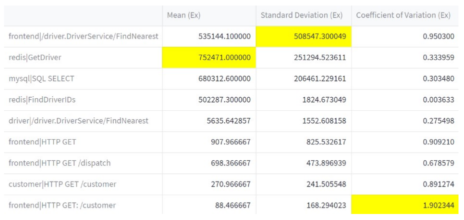
Performance on Scenario 3:

Scenario 3 differs from the cases that were tested before in that two artificial latencies were injected into two separate span: in our case, `redis|GetDriver` as well as `mySQL|SQL Select`. The algorithm did correctly highlight `redis|GetDriver`, but had issues correctly pointing out `mySQL|SQL Select`, as shown in Figure 12. In the column of coefficient of variation of exclusive latency, the span `frontend|HTTP GET: /customer` was highlighted. With Figure 13, we can notice that that this span is actually a relatively nearby parent span of `mySQL|SQL Select`. Although the algorithm does not highlight all high-uncertainty spans in the case where there are more than one latency issue happening, it is still able to locate a nearby parent span, greatly shrinking the potential area for trace point placement.

As can be seen from Figure 14, the bimodality of `redis|GetDriver` was again accurately picked up by our tool.

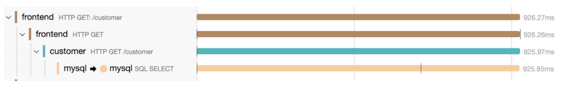
Performance on Scenario 4:

Scenarios 3 and 4 share the same latency issues, which our algorithm picked up here in Scenario 4



	Mean (Ex)	Standard Deviation (Ex)	Coefficient of Variation (Ex)
frontend|/driver.DriverService/FindNearest	535144.100000	508547.300049	0.950300
redis|GetDriver	752471.000000	251294.523611	0.333959
mysql|SQL SELECT	680312.600000	206461.229161	0.303480
redis|FindDriverIDs	502287.300000	1824.673049	0.003633
driver|/driver.DriverService/FindNearest	5635.642857	1552.608158	0.275498
frontend|HTTP GET	907.966667	825.532617	0.909210
frontend|HTTP GET /dispatch	698.366667	473.896939	0.678579
customer|HTTP GET /customer	270.966667	241.505548	0.891274
frontend|HTTP GET: /customer	88.466667	168.294023	1.902344

Figure 12. Uncertainty table of spans for Scenario 3.



frontend HTTP GET: /customer 926.27ms
frontend HTTP GET 926.26ms
customer HTTP GET /customer 925.97ms
mysql ➡ mysql SQL SELECT 925.85ms

Figure 13. A section from a trace, shown in Jaeger, displaying the nearby parent spans of `mySQL|SQL Select`

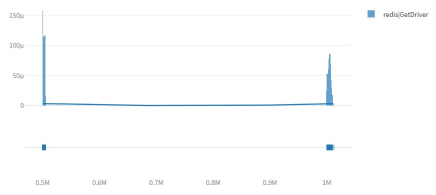
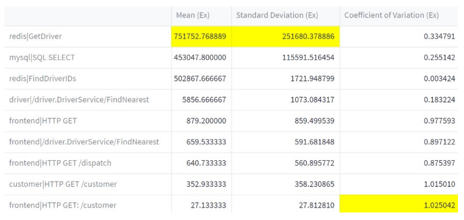


Figure 14. Histogram of latency for `redis|GetDriver` in Scenario 3.

with a fashion similar to how it processed Scenario 3: as shown in Figure 15, `redis|GetDriver` was highlighted, while a nearby parent span of `mySQL|SQL Select` was also pointed out. However, our algorithm was not able to pick up on the critical difference between Scenarios 3 and 4 – that is, the fact that the two latency issues in Scenario 4 were dependent on each other.



	Mean (Ex)	Standard Deviation (Ex)	Coefficient of Variation (Ex)
redis|GetDriver	751752.768889	251680.378886	0.334791
mysql|SQL SELECT	453047.800000	115591.516454	0.255142
redis|FindDriverIDs	502867.666667	1721.948799	0.003424
driver|/driver.DriverService/FindNearest	5856.666667	1073.084317	0.183224
frontend|HTTP GET	879.200000	859.499539	0.977593
frontend|/driver.DriverService/FindNearest	659.533333	591.681848	0.897122
frontend|HTTP GET /dispatch	640.733333	560.895772	0.875397
customer|HTTP GET /customer	352.933333	358.230865	1.015010
frontend|HTTP GET: /customer	27.133333	27.812810	1.025042

Figure 15. Uncertainty table of spans for Scenario 4.

Once more, as seen in Figure 15, the bimodality of `redis|GetDriver` was accurately picked up by our tool.

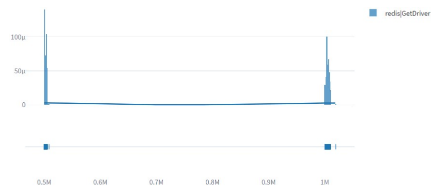


Figure 16. Histogram of latency for `redis|GetDriver` in Scenario 4.

We can confirm that the artificial bimodality that existed in Scenarios 1, 3, and 4 were indeed artificial and does not stem from HotROD itself by observing the histogram of the latencies of `redis|GetDriver` in HotROD without any artificial latencies. Figure 17 presents this information. While it shows a general bimodal trend, the peaks are each much wider than those show in the previous histograms, therefore still presenting a very different structure than any of the artificially-injected latencies have made.

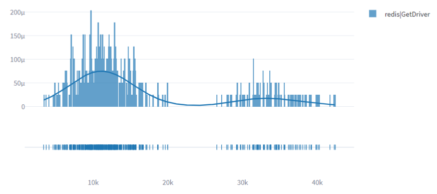


Figure 17. Histogram of latencies for `redis|GetDriver` in the Control Scenario.

5 Limitations and Future Work

While our algorithm is able to straightforwardly identify latencies, it operates based on a key major assumption about spans that is not necessarily realistic in some real life services that are based on distributed systems. Our algorithm assumes that while a parent span is waiting on child span, it does not do anything itself besides potentially calling other children spans. This assumption greatly simplifies the complexity of the system's dynamic, but can most certainly contributes to the oversimplification of the issue in such systems, especially those that occur on parent spans as they are waiting for their children spans to finish.

The second limitation of our algorithm is that it currently does not support synchronous analysis of distributed systems – analyzing the system as it is currently running and adapting to the behaviour of the system's real-time changes. Our tool requires an input of at least 30 traces for results to be statistically significant, and only supports the analysis of existing JSON files, which can decrease its capability to accurately locate areas of uncertainty for systems that are highly dynamic.

A third limitation of our algorithm is that although it is meant to be used for any given distributed system, it was tested only on HotROD, a simulated platform that can be regarded as relatively simple. HotROD has less than 20 services in total, and the way each span connects to its parent and children spans is extremely straightforward. This limits the variations of latencies issues that it supports, therefore, we were not able to test an extensive set of differing latency issues on our algorithm.

Additionally, there are further limitations that narrow the implementation choices of our program. For instance, the program is unlikely to be particularly useful if its decisions are made solely on the basis of request paths generated by user requests. This is due to the fact that user requests as well as traffic during user requests may vary greatly, resulting in inconsistencies that may confound the program. Thus, there must likely be some component of the program that can also generate and test request paths artificially.

Another issue is that, besides measures of overhead, there is currently no conventionally-used metric for the evaluation of the efficacy of a particular tracing scheme that does not involve surveying human developers. Since this efficacy varies from developer to developer and codebase to codebase (accounting for the different architecture types and API standards that the program may encounter), establishing an objective metric in a universally applicable manner may be difficult.

Another limitation of our algorithm is the fact that it only works at its best when the latency issue(s) in the system that is being analyzed is pronounced. When we conducted tests on our four sets of artificial latencies, we repeatedly changed the length of the latencies in each set by factors of 10 and 100. Our tool worked well in all repetitions, no matter the size of the latencies that were being injected. However, our tool's accuracy only drastically increased as the length of the latencies were multiplied by factors of 100 or more.

In seeking to address these issues, we have considered numerous avenues by which to potentially expand upon our existing prototype. Foremost, in addition to assisting developers by examining latency anomalies, we could consider structural anomalies as well. This would be particularly useful to address cases where structural issues impact end-to-end latency, even where they do not affect individual span latencies.

Another possibility we could consider is alternative methods of representing trace data for processing. For instance, the event-based model (as opposed to the span-based model) would operate on the basis of distinct trace points as opposed to span-based intervals. This model can be useful to reduce ambiguity that the span-RPC model may introduce, such as unclear temporal relationships in parent-child relationships. This would allow spans to contain logs rather than having to rely on internal spans.

5.1 TrainTicket

HotROD is a rather simplistic model of a distributed system. More complex models, such as TrainTicket, may provide better simulations of trace data and issues that a distributed system may face.

TrainTicket is a mostly Java-based simulation of a train booking service based on an architecture with 41 microservices — much more than the 4 implemented in HotROD. Like HotROD, TrainTicket was also programmed to support span visualization with Jaeger, enabling us to insert, delete, and test spans as well as logs like we were able to with TrainTicker. However, what makes TrainTicket an even better alternative to HotROD is the fact that it is written in Java, instead of Golang – the much less used and therefore supported language.

5.2 EBPF

One possibility for the project is not merely to advise developers on the placement of trace points, but additionally to insert trace points as well. This could both automate instrumentation and be used to test how well different placements improve predictability of the request path latency.

This could be approached using a tool called eBPF (extended BPF). It is a tool that works quite analogously to a virtual machine that is able to provide users access to the memory of the Linux kernel. The most important part of eBPF, though, lies in its "probes." BPF (and, in extension, eBPF) provides users with probes that they are able to insert in any place in a program. These probes are then able to collect and transmit information around the environment in which this specific piece of code was executed. For example, one type of probes are uprobes (userspace probes), which users can stick to any userspace program for it to inject interrupting instruction so that it can capture the arguments of the function it is in. This technique is also the core logic of how many debuggers with trace points work. eBPF also provides to its users many other types of probes that are suited to record information in a wide variety of programs, like kprobes that collect information from the kernel. With its spectacular ability to hook probes and record information, as well as its support for many languages (including Golang and Java), eBPF is one of the first testing techniques that we will be trying to implement in the next stages of our project.

5.3 Machine Learning

Another tool that we have considered in machine learning. This is a tool that could work perfectly for a complex task such as ours. For instance, a machine learning algorithm could be used to learn the intricacies and standards of a particular distributed system as well as learn the edge cases and errors, rendering it an adaptable tool. The parallels between this task and the applications of machine learning were also identified previously by the creators of Sifter [3] for more details).

However, we have largely set the idea of machine learning aside due to the issue of practicality. Developing a machine learning model for our task would require large amounts of training data. While the developers of Sifter [3] used simulated tracing data to train the program, this is often impractical as it does not accurately simulate the request paths of real distributed systems.

5.4 Content

Similarly to the issue with placement discussed in 2.4, if instrumentation neglects the content of trace points, the developer may only receive temporal information on the latency between servers in the system whereas the true cause of a bottleneck may instead be restricted memory; or, the developer may instrument trace points to record every metric available, resulting in once again needing to later sift through extraneous data to determine the true cause of an anomaly. On the other hand, if the developer has an abundance of irrelevant information regarding content stored in each trace point, it will again be difficult to differentiate the useful ones as the developer has to manually run through a big volume of data to locate them. Recording too much information down will also inevitably create big overhead and slow down the program by taking up too much memory.

The ideal content that a trace point should contain is dependent on the performance issues that often appear, which can change throughout different sections of the code. For example, a performance issue can be "workload not getting the resources it needs to complete in time" or "the resource is obtained but is not fast enough to provide the desired response time." That is, "content" deals with problems such as resource usage, span genealogy, and metrics. Some examples are:

- The address spaces that have the highest delays in the system
- Important resources as well as who is using and who is waiting for these resources
- Information about storage consumption, like paging, migration, frames available, etc.
- The utilization of common storage
- Activities in the CPU, cache system, and RAM

Looking at options of reports like the ones above, we know that we need trace points to return information like the address spaces that have high delays, and, connecting to that, which part of the application is using this address space, as well as which parts of application is currently waiting for this address space to become empty and is thus being delayed.

Currently, our prototype does not utilize span contents, like internal logs or host information. Rather, we rely upon span names to group behaviors. In the future, we could additionally use internal logs or key-value pairs if they are included in the spans to reduce uncertainty without needing to add additional instrumentation. As such, advising developers on content could be an intriguing avenue for our prototype.

6 Conclusion

Developers of distributed systems use distributed tracing to provide visibility into the complex interactions within the system. Oftentimes, they must manually instrument their codebases and have difficulty determining where to place instrumentation to maximize utility. We presented a tool that utilizes aggregate trace data from a distributed system to localize regions of high relative uncertainty by ranking services according to uncertainty metrics, like the standard deviation of exclusive latency. Our tool is capable of accurately uncovering and highlighting the services with the highest uncertainty within a system, no matter if the issue causing the uncertainty acts within traces or from one trace to another. In addition, our tool is also able to accurately pick out and visualize significant patterns within the distribution of exclusive latencies of the services within a system. To further continue our research, we hope to take measures such as implementing other statistical uncertainty measures or incorporating other tracing implementation tools like eBPF.

7 Acknowledgements

We would like to thank the MIT PRIMES Computer Science Program, our PRIMES mentors Dr. Raja Sambasivan, Darby Huye, Max Liu, and Zhaoqi Zhang for guiding us and giving us this opportunity to conduct research and learn as a part of this amazing program.

References

[1] Spans. URL `https://opentracing.io/docs/overview/spans/`.

[2] Jaeger: open source, end-to-end distributed tracing. URL `https://www.jaegertracing.io/`.

[3] Pedro Las-Casas et al. Sifter: Scalable sampling for distributed traces, without feature engineering. Association for Computing Machinery, Nov 2019. doi:https://doi.org/10.1145/3357223.3362736.

[4] Mert Toslali et al. Automating instrumentation choices for performance problems in distributed applications with vaif. Association for Computing Machinery, Nov 2021. doi:https://doi.org/10.1145/3472883.3487000.

[5] Xiaofeng Guo et al. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. Association for Computing Machinery, Nov 2020. doi:https://doi.org/10.1145/3368089.3417066.

[6] Milind Chabbi and M.K. Ramanathan. Crisp: Critical path analysis for microservice architectures. November 2021. URL `https://www.uber.com/blog/crisp-critical-path-analysis-for-microservice-ar`

[7] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

[8] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow`.

[9] Jonathan Leavitt. Duckee go: Dynamic and user-friendly concolik execution engine in go. May 2014.

[10] Data model — openzipkin. URL `https://zipkin.io/pages/data_model.html`.

[11] Benjamin H. Sigelman et al. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010. URL `https://research.google.com/archive/papers/dapper-2010-1.pdf`.

[12] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, Apr 2007. USENIX Association. URL `https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework`.

[13] Dénes Vadász. Open for event based tracing? Jan 2019. URL `https://medium.com/opentracing/open-for-event-based-tracing-a326c295f2a2`.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi:https://doi.org/10.1145/359545.359563.