Massachusetts Institute of Technology Program for Research in Mathematics, Engineering and Science for High School Students (MIT PRIMES 2021)

Threshold-Based Inference of Dependencies in Distributed Systems

Gupta, Tanmay and Rastogi, Anshul

MIT PRIMES 2021
Mentors: Darby Huye, Max Liu, Dr. Raja Sambasivan

tanmaygupta2023@gmail.com, stellaranshul@gmail.com

Abstract -

Many current online services rely on the interaction between different components that form a distributed system. Analyzing distributed systems is important in performance analysis (e.g. critical path analysis), debugging, and testing new features [1, 2]. However, the analysis of these systems can be difficult due to limited knowledge of how components work and the variety of services and applications that are usually instrumented. *The Mystery Machine* , introduced by Chow et al. in 2014, has a "big data" approach, using logged events across many traces to generate and refine a causal model [1]. We introduce *Scooby Systems*, our extension of *The Mystery Machine*'s algorithm. We introduce thresholds to increase the tolerance to violations in the formation of causal relationships. In the future, we hope to improve *Scooby Systems*'s scalability with a Hadoop MapReduce implementation.

Keywords -

Causal Model; *The Mystery Machine*; Big Data; Distributed Debugging; Distributed Tracing

1 Introduction

1.1 Problem

Distributed systems are essential to how the modern world operates. However, the maintenance of distributed systems can be riddled with obstacles. Specifically, the structure of large distributed systems may be difficult to understand due to black box components and numerous distinct services or applications interacting with one another. Distilling causal relationships between components of the system from trace data is important for the comparison of different deployments of these systems, the identification of latency or structural anomalies, or the elucidation of request paths to facilitate the debugging, optimization, and modification of the system.

While numerous software exist for tracing distributed systems (detailed in Section 1.2), many require extensive instrumentation of the system to operate, and many do not generate a comprehensive causal model of the system. However, *The Mystery Machine,* introduced in 2014, requires less instrumentation and infers the system's structure based on trace data from the system's logs [1].

Nevertheless, *The Mystery Machine,* while possibly more flexible, also bears a host of limitations. For instance, *The Mystery Machine*'s algorithm in particular is intolerant to inaccuracies in inputted logs; a single error in the provided data can result in an erroneous output model. This rigidity means that issues in timestamps caused by clock skew or missing logs can result in an erroneous output.

We wished to replicate and then expand upon *The Mystery Machine*'s "big data" approach to allow for a user-controlled probability threshold for dependencies to provide a tolerance to timestamp errors caused in traces by clock skew or anomalies.

1.2 Background

There are several methods that software programs currently use for end-to-end tracing [3].

As examples, we will discuss Dapper's and X-Trace's approaches to tracing. Dapper uses metadata propagation (such as the propagation of span parent IDs) to capture caller-callee relationships [2]. However, Dapper fails to capture all causal relationships; we discuss this in detail in Section 1.4. X-Trace can capture a more comprehensive collection of causal relationships between system components. However, it relies on developers' application-specific knowledge to instrument the system and specify causal relationships between components [4].

However, these tracing frameworks that rely heavily on application-specific knowledge can be difficult to implement in heterogeneous systems whose components are very diverse (e.g. many different programming languages present). It can often be extremely time-consuming and difficult to instrument components of such distributed systems, such as "client machines" [1].

The Mystery Machine , a "big data" approach introduced in 2014, uses logged events across multiple requests to infer a similar level of comprehensive causal relationships as X-Trace, but with the benefit of requiring much less instrumentation of the system that relies on application-specific knowledge. Nevertheless, even though *The Mystery Machine* uses logged events instead of the instrumentation of communication points, it still requires the logs to contain a "minimal schema" of information, such as the request ID [1]. In this paper, we focus on extending *The Mystery Machine*'s algorithm, employed by Chow et al. [1].

1.3 The Ideal Trace

Definition 1.3.1 (Span)**.** "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" [5] whose boundaries are determined by system calls.

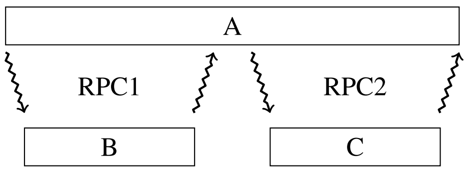

Figure 1. A trace represented with the span model

In Figure 1, span A calls spans B and C (these calls are shown as RPCs, represented by zigzagging arrows throughout the paper). Note that time increases from left to right in all span-based trace figures in this paper.

The ordering of span B and span C in this trace could be purely incidental — based on this information, we do not know whether or not span C could occur before span B in another request (see Section 1.4 for more detail). In an ideal trace, we want to unambiguously display the causal relationships present in the system.

Thus, in our version of the ideal trace, for various reasons (some of which are detailed in Section 4.1), we choose to use events instead of spans. For instance, we denote the start of span A as an event A_S and the end of span A as A_E.

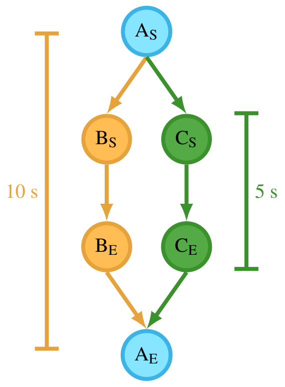

Figure 2. The Ideal Trace

What would the ideal trace of a system resemble? One possibility is demonstrated in Figure 2. This is an idealized representation of the example of a simple distributed system comprised of spans A, B, and C displayed in Figure 1. Immediately, we can discern several things from Figure 2.

To begin, every arrow in the figure represents a *happens-before relationship* as defined in Definition 1.3.2.

Definition 1.3.2 (Happens-Before Relationship)**.** If two intervals of time (such as segments or spans) or points in time (such as events) X and Y exist such that Y starts or occurs strictly after X ends or occurs, we say that X and Y have a *happens-before* relationship, denoted as X→Y in text and as a straight, directed arrow from X to Y on a diagram.

Additionally, we define *concurrent relationships*, which are also displayed in the ideal trace.

Definition 1.3.3 (Concurrent Relationship)**.** Two intervals or points in time in a causal structure are said to be *concurrent* if both can occur independently of one another.

We see that the events of span B (B_S and B_E) have no happens-before relationships to any of the events that define span C (C_S and C_E). Therefore, this trace tells us immediately that spans B and C are concurrent. Concurrency, as seen in the figure, is represented by a "fan-out."

Additionally, using the ideal trace, we can identify the longest path (critical path) and observe that it passes through span B. Therefore, to optimize this system, a developer would focus on optimizing the tasks that occur as a part of span B.

This ideal trace, therefore, provides abundant information about this simple system in an unambiguous manner.

1.4 Tracing: The Challenges

The Mystery Machine's use of temporal data to form causal relationships contrasts other tracing frameworks, such as those in systems like Dapper and Jaeger. Causal relationships in these frameworks can be indicated with spans and caller-callee relationships (Note: we use RPC calls as an example of calls between parent and child spans in the remainder of the paper). For example, Jaeger can display spans in a Gantt chart, which presents temporal information and parent-child RPC relationships between spans [6].

Consider, for instance, the simple structure demonstrated in Figure 3.

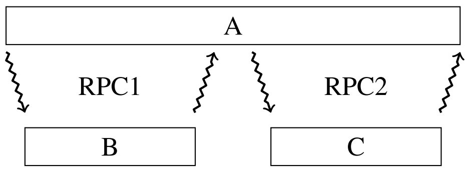

Figure 3. RPC relationships between spans A, B, and C

In Figure 3, RPCs are represented with zigzagging arrows. For example, the call from A to B in RPC 1 shows that span A is span B's parent span. Similarly, the call from A to C in RPC 2 shows that span A is also span C's parent span. While this model shows developers caller-callee relationships, it does not capture all of the system dependencies.

Using the temporal data of this single trace in Figure 3, we can observe that span C occurs after span B. However, we have no way of knowing if this is due to span C being dependent on span B or if the two child spans of span A can execute concurrently. In other words, given just this trace information, one cannot infer whether or not the information from RPC 1 must return to span A before the execution of RPC 2. For instance, if span B returns a value to span A that is then manipulated and passed as a parameter to span C, there would be a dependency between spans B and C; span C's initiation would be stipulate span B's completion. Conversely, it is possible that span A was calling multiple other children spans, resulting in span B being scheduled before span C in this particular request despite there being no requirement of span B completing before span C — that is, in another request, span C could be scheduled before span B without affecting the function of the request.

The inability of the model in Figure 3 to convey the precise network of dependencies results in an incomplete image of the structure for developers that may not be familiar with the particular service or application being traced. Thus, the tracing frameworks that use such a model may rely on individual programmers' knowledge of the various applications in the distributed system to understand the dependencies not shown by the data or to explicitly provide that information to the tracing software.

Instead, *The Mystery Machine*'s approach — using temporal data across many traces, instead of just a single trace — mitigates the issue of relying on programmers' knowledge of their applications. This is explained further in Subsection 2.2.

2 Mystery Machine

2.1 The Big Data Approach

The Mystery Machine approaches the ideal trace using a unique "big data" method — displaying causality within a distributed system using logged events over an enormous sample of requests as opposed to using a single request.

To begin, we must understand how *The Mystery Machine* processes log data. *The Mystery Machine* constructs its model from logged events in a request, which then form the endpoints of temporal intervals denoted *segments* [1].

Definition 2.1.1 (Segments)**.** Chow et al. define trace *segments* as "the execution interval between two consecutive logged events for the same task," where a task is a "distributed thread of control." [1].

We further discuss segments (as compared to spans and events) in 4.1.

As segments are created from logged events, *The Mystery Machine* operates primarily off of temporal data. Therefore, the causal relationships in the models *The Mystery Machine* forms are based solely on temporal information.

To explore how this enhances *The Mystery Machine*'s functionality in contrast to other tracing frameworks, we depend on the concept of happens-before and concurrency relationships (see Definitions 1.3.2 and 1.3.3).

2.2 *The Mystery Machine*'s Algorithm

We discuss the details *The Mystery Machine*'s algorithm here.

The Mystery Machine begins by iterating through all the traces formed by the logged events of requests. When *The Mystery Machine* encounters a new segment, it forms edges from that segment to all other segments already in the model. That is, the algorithm assumes that all possible edges from and to a newly encountered node exist, where nodes are segments and edges represent happens-before relationships (although it also considers mutually exclusive and pipeline relationships, these are unimportant to our goal and thus are not discussed here). *The Mystery Machine* then refines this causal network as it iterates through the trace data in the following manner: if it encounters a counterexample/violation to one of the hypothetical edges, then the algorithm removes that edge from the causal model. We call this final causal model the distributed system's *global causal model (GCM)* as defined in Definition 2.2.1.

Definition 2.2.1 (Global Causal Model (GCM))**.** The global causal model (GCM) of a distributed system is defined to be a model representative of causal relationships in that system with a scope that cannot be captured solely based on information from one trace.

We now consider a simple example of *The Mystery Machine*'s algorithm. The example consists of the same distributed system as in Figure 3 in which current tracing frameworks were unable to determine the causal relationship between B and C. Figure 4 and Figure 5 show traces (that were constructed from logged events) of two different requests in the system. For simplicity, we only consider the relationships between segments B and C, and we ignore all other segments. *The Mystery Machine*'s algorithm will go through both of the traces. When updating the causal model in Figure 7's trace, the following occurs:

The Mystery Machine begins iterating through the traces. As it encounters B and C in Trace 1 (Figure 4),

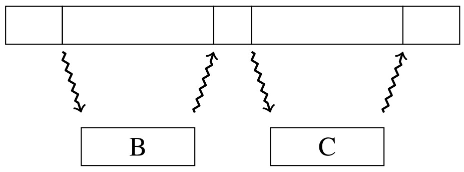

Figure 4. Trace 1 — Segment-based representation of a request

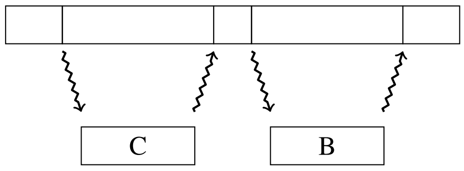

Figure 5. Trace 2 — Segment-based representation of a second request

creates both B→C and C→B in its GCM. Then, as *The Mystery Machine* processes the trace, it sees B→C; this contradicts C→B, so *The Mystery Machine* removes that edge from the GCM. Then, in the second trace (Figure 5), *The Mystery Machine* likewise removes B→C as it sees C→B in the trace. Therefore, the GCM would declare B and C concurrent.

2.3 Reformatted Traces

The Mystery Machine is able to transitively reduce their traces and calculate the critical path, as well as other conduct other analysis, on a "per-request basis," which we refer to as reformatted traces (see Definition 2.3.1).

Definition 2.3.1. Given a trace, we define the reformatted trace as a subset of the global causal model. The reformatted trace contains only events that are present in the original trace (and the relationships between those events that are in the global causal model).

Indeed, reformatted traces would be essentially be "ideal traces" as introduced in Section 1.3 — individual traces represented with accurate causal relationships. For instance, if two concurrent tasks incidentally appear sequential, the ideal trace would still show them as being concurrent, rather than focusing on the temporal order.

2.4 Transitive Reduction in *The Mystery Machine*

We consider the GCM produced by *The Mystery Machine*'s algorithm. If all events are present in every trace, then A→B and B→C would mean that those relationships hold in all traces (since one counterexample is enough to remove an edge). Hence, we can conclude that there are no violations of A→C, and thus, that edge would be left in the global causal model. However, the restriction that all events are present in every trace is not always true. For example, suppose we have two traces. In the first trace, suppose the only relationship found is A→B. In the second trace, suppose the only relationship found is B→C, then the causal model would have edges A→B and B→C, but it would not have any edge A→C (because events A and C never even occur in the same trace). In fact, it could even be the case that in a third trace, C→A. Therefore, applying a transitive reduction to the global causal model does not make sense as the transitivity of the → relation does not hold.

Nevertheless, *The Mystery Machine* is able to transitively reduce their traces on a "per-request basis" [1], which we refer to as reformatted traces (see Definition 2.3.1). This is due to the idea that if events A, B, and C occur together in a specific trace, and A→B and B→C occur in the causal model, then A→B and B→C must occur in the specific trace as well. Then, it follows that A→C in the reformatted trace. Therefore, transitivity holds within the reformatted trace, and transitive reduction is allowed.

3 Limitations of *The Mystery Machine*

The Mystery Machine makes a multitude of assumptions about its trace data to ensure that its algorithm is accurate. These assumptions create several limitations in its design.

The Mystery Machine uses data from *ÜberTrace* , which needs to account for clock skew in the timestamps [1]. This requires having data about RPC communication.

Finally, we list one of the main assumptions on which *The Mystery Machine* relies:

True causality is very hard to determine, but through the temporal data, one can determine happens-before relationships. *The Mystery Machine* assumes that the traces have enough variation that the happens-before relationships left in the final causal model are the only true causal dependencies.

3.1 Repeated Event Names

The Mystery Machine states that "*ÜberTrace* requires that each <event, task> tuple is unique, which implies that there are no cycles that would cause a tuple to appear multiple times" [1]. Here, an event refers to a logged event while a task is defined as where a task is a "distributed thread of control" [1]. Thus, *The Mystery Machine* assumes that events never repeat on the same task.

However, this is neither necessarily true nor generally a safe assumption.

Let us consider two events that both have the same name represented by X. For instance, we can consider an authentication function that is called twice, the second time being after a user enters a wrong password. This could potentially result in events with identical names produced as the same authentication function is re-run, thus causing us to get many unreasonable relationships. For example, we could get X→X, which does not make practical sense.

We propose an idea to mitigate this issue in Section 5.2.

3.2 Excessive Rigidity

Another limitation of *The Mystery Machine* stems from its excessive rigidity. As per its algorithm, even one counterexample to a hypothetical edge among hundreds of thousands of traces will result in that edge being entirely removed from the GCM. Thus, *The Mystery Machine* is inflexible to any errors in the input data — potential for error from an anomalous trace are not accounted for, and *The Mystery Machine* attempts to account for clock skew assuming that RTT (round-trip time — the estimated times for request and response times) is symmetric [1]. We hope to address this rigidity by introducing the concept of a threshold for every edge as detailed in Section 4.2.

4 *Scooby Systems*

Scooby Systems is our extension of *The Mystery Machine*'s "big data" approach to learning causality between events in a distributed system. We discuss some of the choices we made to account for our input data described in Section 4.4 and to improve *The Mystery Machine*'s functionality. This involved addressing some of the limitations of *The Mystery Machine* detailed in Section 3 — in particular, *The Mystery Machine*'s rigidity.

4.1 Spans, Segments, and Events: Trace Structure Interpretations

Before we discuss our algorithm and pertaining concepts, we discuss our choice to represent *Scooby Systems*'s GCM as an event-based model.

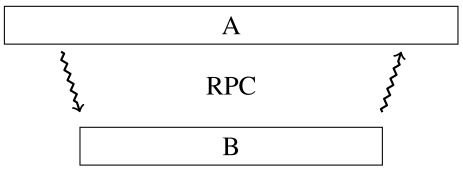

Figure 6. Causal relationship represented with spans A and B

Figure 6 displays an RPC between span A and span B, where span A initiates span B through an RPC. Spans are defined in Definition 1.3.1 and are currently a widespread method of displaying trace structure across tracing algorithms [5].

Spans are effective in displaying caller-callee relationships as users can infer these relationships at a glance to understand the flow of information in a distributed system. We can see in Figure 6 above that span A initiates an RPC that is sent to span B. This means that the execution of span B is dependent on that of span A; span B cannot occur without span A to initiate it.

We now consider another method to represent trace data — segments. Consider the following alternative interpretation of the structure in Figure 7:

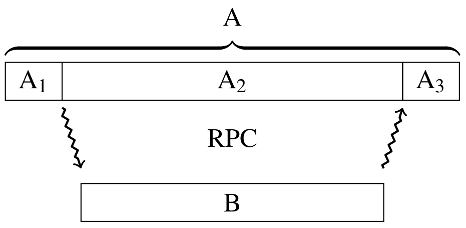

Figure 7. Trace Segments

We see that span A above is divided into smaller intervals where span A sends and receives the RPC to span B. These smaller intervals are precisely what we refer to as *segments*, as defined in Definition 2.1.1.

In the example above, we assume that RPC send-receive instances are logged events. This is not always the case, as is the case with our input data (see Section 4.4).

Figure 8 shows Figure 7's trace structure would be represented with a segment-based causal model. We note the following:

- Segment B and Segment A_2 are dependent on Segment A_1
- Segment B and Segment A_2 are concurrent
- Segment A_3 is dependent on Segment B and Segment A_2

Using these observations to make causal relationships yields Figure 8.

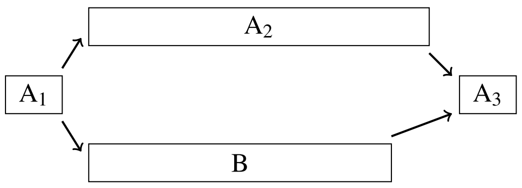

Figure 8. Trace Segments: Causal Representation

We see in Figure 8 that the concurrency between B and A_2 is clear as there is no happens-before relationship between the two. The fact that the segment model lends to these causal relationships makes them preferable for developers. Spans do not have the same flexibility; they rely on caller-callee relationships to convey a limited picture of causality. Spans are better designed to portray duration [7] and less suitable for causal models.

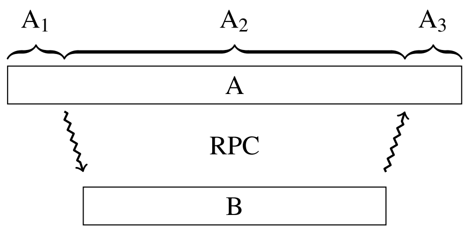

Figure 9. Span Failure

This is most clear when contrasting figures 8 and 9. In the former, segments allow for a clear inference of a causal temporal relationship by means of happens-before.

However, displaying causality with spans can be fundamentally impossible. For example, parts of span A are concurrent with span B but other parts are not. A single caller-callee relationship between the two spans cannot capture these more nuanced relationships.

We now discuss events as a method of interpreting trace structure. Consider Figure 7 with the event annotations E_i superimposed with the trace segments.

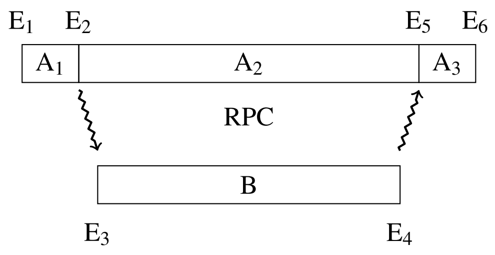

Figure 10. Events (denoted E_i) added to Figure 7

This, in a sense, is what algorithms such as *The Mystery Machine*'s receive; these logged events are translated as the ends of the aforementioned segments. However, the events and the segments differ in displaying causal relationships; we can see this with the example from Figure 11, which represents the happens-before relationships translated from Figure 10.

We see from Figure 11 that the event-based representation is related to the segment model. If using the event model in Figure 11, the user would would have to see that E_2 and E_5 are start and end of A_2 and they encompass start and end of B, which are E_3 and E_4. Then, the user would have to apply *The Mystery Machine*'s definition of happens-before to A_2 and B to conclude that the two are concurrent.

We believe that this lengthier process — the identification and comparison of two pairs of events — reaches the same conclusion that the segment-based model in Figure 8 would give us in a single glance. Therefore, while events are quite accurate in a depiction of causal structure, they can also be cumbersome for a user to interpret.

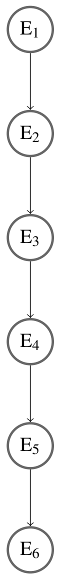

Figure 11. Event-based Causal Representation

However, events have one other advantage that segments lack; they are tolerant of missing data. Whereas the accuracy of the segment model requires logged events for when RPC messages are sent and received, the event model does not. For instance, the event model could operate solely off of span start and end events (as *Scooby Systems* does) whereas the segment model would collapse.

This is precisely why we must resort to the event model. In the trace data we used, the spans did not have logged events (as detailed in Figure 4.4), thus preventing us from splitting spans into segments. *The Mystery Machine* assumes that the given data provides the necessary logged events to construct an accurate output — it "cannot identify causal relationships involving the unlogged segment" [1]. Therefore, our best model is the event-based interpretation of trace structure.

4.2 User-Defined Threshold

We extend beyond *The Mystery Machine* by employing a threshold that increases the algorithm's tolerance to flawed data. Only edges that pass the condition provided by threshold are displayed in the output model.

The threshold value is user-defined, allowing the user to customize the functionality of the program as they desire. For instance, if the user expects there to be no mistakes in the provided data, they can choose the route of *The Mystery Machine* and set a threshold of 100% for the number of successes of an edge out of the total number of traces (see Subsection 4.2 for specific definitions and more details).

The definition of *threshold* depends on the concept of an edge's *successes*, *violations*, and *unknowns*, so we define those terms first.

Definition 4.2.1 (Timestamp)**.** The *timestamp* of an event A is denoted by t_A. We assume that this *timestamp* is given as a quantity of some arbitrary temporal unit, such as nanoseconds or microseconds.

Definition 4.2.2 (Success (of an edge))**.** A *success* of an edge $A \longrightarrow B$ is a instance where events A and B both occur in a trace and $t_A < t_B$ in that trace. The number of traces with *successes* of an edge $A \longrightarrow B$ is denoted as $s_{A\rightarrow B}$.

Definition 4.2.3 (Violation (of an edge))**.** A counterexample of an edge $A \longrightarrow B$ is considered a *violation* for the edge. Specifically, if $t_A \geq t_B$ in a trace, it is a *violation* of $A \longrightarrow B$. The number of traces with *violations* of an edge $A \longrightarrow B$ is denoted as $v_{A\rightarrow B}$.

Definition 4.2.4 (Unknowns (of an edge))**.** If in a trace, at least one of A or B does not occur at all, we call this an *unknown* of $A \longrightarrow B$ and $B \longrightarrow A$. The number of traces with unknowns of an edge $A \longrightarrow B$ is denoted by $u_{A\rightarrow B}$. Note that these unknown relationships can occur when some components of a distributed system are not used in processing a certain request.

Definition 4.2.5 (Threshold Condition)**.** The *threshold condition* for an edge $A \longrightarrow B$ is an inequality using the variables $s_{A\rightarrow B}, v_{A\rightarrow B}$, and $u_{A\rightarrow B}$. The edge $A \longrightarrow B$ is only included in the global causal model if the threshold condition is true for that edge.

There is not just one way to define a threshold. For example, one might want to look at the proportion of traces in which the edge has a success out of the total number of traces T. So, we could have $\frac{s_{A\rightarrow B}}{T} \geq c$ for some threshold frequency c. Alternatively, we could count unknowns as well, so we would have $\frac{s_{A\rightarrow B}+u_{A\rightarrow B}}{T} \geq c$.

Note that considering $\frac{s_{A\rightarrow B}+u_{A\rightarrow B}}{T} \geq c$ is the same as $\frac{v_{A\rightarrow B}}{T} \leq 1-c$.

If we consider *The Mystery Machine* , it defines an edge $A \longrightarrow B$ to be valid only if $\frac{v_{A\rightarrow B}}{T} \leq 0$, which is equivalent to saying $\frac{s_{A\rightarrow B}+u_{A\rightarrow B}}{T} \geq 1$. There are also other options, such as looking at the ratio between successes and violations. For a given user, they could prefer one definition of the threshold based on certain properties of their system. For example, if a certain system is being updated and has many new events occurring, one might want to use the threshold where $s_{A\rightarrow B}$ and $u_{A\rightarrow B}$ are grouped together so that causal model does not assume events from new features were not originally dependent on the events previously present in the model. Thus, due to the variety of choices for defining a threshold, we let the user choose their threshold.

4.3 Subtractive and Additive Models

Definition 4.3.1 (Subtractive Model)**.** We call GCM-producing algorithms for constructing causal models *subtractive* if they track the occurrences of violations or counterexamples of edges to determine if each edge passes the threshold and updates the causal model accordingly.

The Mystery Machine employs a *subtractive* algorithm since makes all changes to its GCM based on violations. However, since *The Mystery Machine* has zero tolerance for violations, it can remove edges as it iterates through the trace data, instead of updating the causal model after iteration.

Definition 4.3.2 (Additive Model)**.** On the other hand, we say GCM-producing algorithms are *additive* if they track the occurrences of successes of edges to determine if each edge passes the threshold and updates the causal model accordingly.

Considering each relationship between two events (e.g. $A \longrightarrow B$), we have 3 possible relationships in each trace: successes, violations, and unknowns.

Now, let us consider the additive model — the GCM-producing data structure will have an element that stores $s_{A\rightarrow B}$. It will also have an element that stores $s_{B\rightarrow A}$.

So, we already have $s_{A\rightarrow B}$.

Now, consider $v_{A\rightarrow B}$. A violation can only happen when $B \longrightarrow A$ or $t_A = t_B$. So, we would need to count the cases when $t_A = t_B$. We call such cases *simultaneous* as defined in Definition 4.3.3.

Definition 4.3.3 (Simultaneous (of events))**.** We say that two events A and B are *simultaneous* if $t_A = t_B$.

Let us denote that with a variable q that represents the number of traces in which t_A is equal to t_B. Thus,

$$v_{A\rightarrow B} = s_{B\rightarrow A} + q.$$

Finally, we compute $u_{A\rightarrow B}$. If we let the total number of traces be T,

$$u_{A\rightarrow B} = T - s_{A\rightarrow B} - v_{A\rightarrow B} = T - s_{A\rightarrow B} - s_{B\rightarrow A} - q.$$

If we choose to implement the subtractive model, it will essentially be counting the number of violations of each edge. So, in the GCM-producing data structure, we will have an element for $v_{A\rightarrow B}$ and $v_{B\rightarrow A}$. We already have $v_{A\rightarrow B}$. Next, we compute $s_{A\rightarrow B}$. Note that $v_{B\rightarrow A}$ means $t_B \leq t_A$. So, if q counts the number of traces where $t_A = t_B$,

$$s_{A\rightarrow B} = v_{B\rightarrow A} - q.$$

Finally, given the total number of traces is T,

$$u_{A\rightarrow B} = T - s_{A\rightarrow B} - v_{A\rightarrow B} = T - v_{B\rightarrow A} - q - v_{A\rightarrow B}.$$

So, it makes sense that *The Mystery Machine* uses the subtractive model, since their threshold depends solely on the number of violations (If there is even one violation, Mystery Machine removes the edge). If we refer to our analysis of the subtractive model, we can see that $v_{A\rightarrow B}$ can be calculated directly, without needing to store q as well.

Given that we want to allow the user-defined formulae for thresholds, we want our causal model to be able to calculate $s_{A\rightarrow B}, v_{A\rightarrow B}$, and $u_{A\rightarrow B}$.

Thus, we observe from our analysis that the additive model and the subtractive model are equivalent in terms of the information they provide us (assuming both models also keep track of cases when $t_A = t_B$.)

There is one other option that we have considered, which is counting both successes and violations for each potential happens-before edge. Then, after iterating through all traces, we have $s_{A\to B}$ and $v_{A\to B}$. Then, we could just calculate $u_{A\to B}$ by doing $T - s_{A\to B} - v_{A\to B}$. While this method works, we end up counting times when $t_A < t_B$ under both $s_{A\to B}$ and $v_{B\to A}$, which may lead to larger numbers in the GCM-producing data structure.

Similarly, in the subtractive model, $t_A = t_B$ is not only counted in q, but it is also counted in $v_{A\to B}$ and $v_{B\to A}$.

Although $t_A = t_B$ is less likely to occur than $t_A < t_B$ or $t_A > t_B$, the additive model does not count cases twice. In particular, $t_A < t_B$ is counted only in $s_{A\to B}$. $t_A = t_B$ is counted only in q. Finally, $t_A > t_B$ is counted only in $s_{B\to A}$.

Thus, we chose to use the additive model in our implementation; an overview of our algorithm can be found in Section 4.5.

4.4 Input Data

Unlike *The Mystery Machine* , we do not assume that each component has preexisting logged events that contain the information specified in the minimal schema. Therefore, we use span-based traces and split the spans into the start and end events to acquire our data.

The traces we used for our testing were in JavaScript Object Notation (JSON). We used JSON parsers to facilitate the use of trace data in our program. An example trace (adapted from a DeathStarBench [8] trace) is shown in Figure 12. It contains one span.

From data such as this, we assign a *nameID* to the span as defined in Definition 4.4.1.

Definition 4.4.1 (nameID (of a span))**.** We denote the nameID of a span to be the string formed by concatenating the span's operationName and serviceName. The serviceName of a span is acquired by finding the serviceName corresponding to the span's processID. As an example, the nameID of the span in Figure 12 would be "UploadUserMentions_user-mention-service", where the span's operationName component is "UploadUserMentions" and the serviceName is "user-mention-service" (this corresponds to the span's processID of "p1" when referenced in the "processes" dictionary in the JSON).

However, for the reasons discussed in 4.1, we must use events instead of trace segments; furthermore, the limitations of our data restrict us to only creating events for the endpoints of spans. These endpoints' timings are determined using the span's startTime and duration: the "start" event occurs at the span's startTime and the "end" event occurs at the startTime added to the duration.

```
{
   "traceID":"944c8368543f21fb",
   "spans":[
      {
         "traceID":"944c8368543f21fb",
         "spanID":"a33138a40cc983cb",
         "flags":1,
         "operationName":"
            UploadUserMentions",
         "references":[
            {
               "refType":"CHILD_OF",
                "traceID":"944
                   c8368543f21fb",
                "spanID":"21
                   bac5cf57ddd3f2"
            }
         ],
         "startTime":1557906198307022,
         "duration":8309,
         "tags":[
            {
               "key":"internal.span.
                  format",
               "type":"string",
               "value":"proto"
            }
         ],
         "logs":[

         ],
         "processID":"p1",
         "warnings":null
      }
   ],
   "processes": {
     "p1": {
        "serviceName":"user-mention-
           service",
            }
        }
     }
   }
}
```

Figure 12. JSON Trace

We identify these events with *eventIDs*, which we define in Definition 4.4.2.

Definition 4.4.2 (eventID (of an event))**.** We create the eventIDs of the events of a span by appending to the span's nameID "_S" and "_E" for "start" and "end" events, respectively. For instance, the event that denotes with start of the span in Figure 12 would be assigned the eventID "UploadUserMentions_user-mention-service_S" while the event denoting the end of this span would be similarly assigned the eventID "UploadUserMentions_user-mention-service_E".

4.5 Our Algorithm

In the implementation of our program, we approach the ideal trace through the use of *The Mystery Machine*'s "big data" approach. Thus, we similarly iterate through multiple traces of requests through a particular distributed system to create a GCM.

Our algorithm consists of three stages: *Preprocessing*, *Processing*, and *Postprocessing*.

Preprocessing involves iterating through all traces to record spans, construct their nameIDs (nameIDs are discussed in Section 4.4), and assign them a *reference index*; a span with the kth new nameID constructed receives a reference index of $k-1$. At the end of the Preprocessing stage, a matrix S and an array Q are constructed to record successes and simultaneous relationships, respectively.

As each span's nameID has an individual reference index and each span produces two events, there are a total of $2n$ events (and therefore eventIDs) produced by the recorded nameIDs (once again, there are a total of n nameIDs). S is an adjacency matrix of all events and is therefore a $2n \times 2n$ matrix.

Likewise, Q is a one-dimensional array. Ordering does not matter in this array, unlike in S; therefore, each pair of events requires only one space. There are $\binom{2n}{2} = n(2n-1)$ such event pairs. Hence, Q is of length $n(2n-1)$.

The algorithm then enters the Processing stage. Once again, it iterates through all the traces. Then, it takes each pair of spans and compares the timestamps of the start and end events of both spans; this process is outlined in more detail in Section 4.6. Accordingly, the algorithm increments S and Q in the appropriate spots using the reference indices. For instance, $S[i][j]$ is increased by 1 whenever a new success is found for $e_i \longrightarrow e_j$, where e_i and e_j are events.

Finally, in the Postprocessing stage, the algorithm applies the threshold condition to the information in S and Q to construct the GCM from potential happens-before relationships.

Algorithm 1: Processing Psuedocode

```
for each trace $\tau$ do
    for each pair of spans $(X, Y)$ such that $X, Y \in \tau$ do
        if timestamp conditions then
            Update $S$ and $Q$ accordingly;
```

4.6 Pairwise Comparison of Spans

We considered a variety of algorithms to update the causal model for each event pair. We decided to compare the timestamps of pairs of spans, for a variety of reasons. One of the main reasons we chose to do this was because we wanted to preserve the information about the start and end events of a span; if we treated the start-end event pairs outside the context of their spans, we would have to make more comparisons, thus making our algorithm less efficient. For instance, consider the start and events of two spans X and Y: X_S, X_E, Y_S, and Y_E. The following are examples of scenarios that allow us to reduce the amount of comparisons required: It is always true that

- $X_S \longrightarrow X_E$
- $Y_S \longrightarrow Y_E$
- If $X_E \longrightarrow Y_S$, we know what all the other relationships between events of the two spans must be.

We made similar optimizations to increase the efficiency of our code. Refer to our code (Subsection 4.7) for more information.

4.7 Code

We have stored our code that is made to run on a single machine at the following GitHub repository: `https://github.com/TanmayGupta23/Threshold-Based-Inference-Of-Dependencies`. We are currently working on a Hadoop implementation of our code. See Section 5.3 for the most up-to-date version of our code.

4.8 Transitive Reduction in Our Implementation

We considered whether or not it would be appropriate to transitively reduce the reformatted traces in our extension of *The Mystery Machine* with the threshold conditions. Due to a phenomenon called *transitivity failure*, we determined that we would not transitively reduce the reformatted traces.

Definition 4.8.1 (Transitivity Failure)**.** *Transitivity failure* occurs when a relationship or edge is falsely transitively implied to pass the threshold due to connected edges that form the particular relationship or edge. For example, if edges (l, m) and (m, n) pass the threshold but (l, n) does not.

For instance, consider a set of four traces with the following happens-before relationships between events A, B, and C with a threshold condition of

$$\frac{s}{T} \geq 0.75.$$

We display the relationships between events in these traces in Figure 13.

We see in Figure 13 that the relationship A $\longrightarrow$ B occurs with frequency 0.75 across the traces, A $\longrightarrow$ C occurs with frequency 0.50, and B $\longrightarrow$ C occurs with frequency 0.75. Therefore, the relationships A $\longrightarrow$ B and B $\longrightarrow$ C both pass the 75% threshold, but A $\longrightarrow$ C, which would transitively implied by A $\longrightarrow$ B and B $\longrightarrow$ C, does not.

Note that in both Figure 14, each edge is labeled with $\frac{s}{T}$ in this figure.

Note that Figure 14 also represents the threshold-based GCM of the traces.

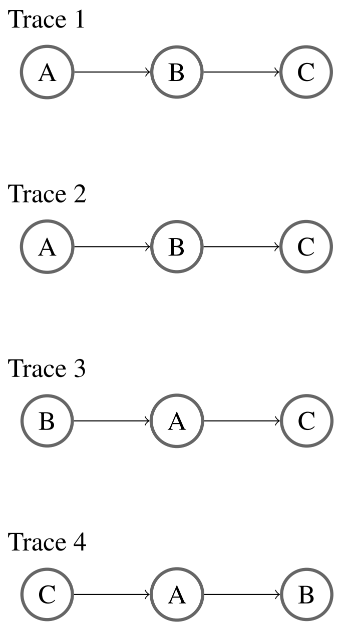

Figure 13. Transitivity Failure Example

As we can see, the information contained in the reformatted Trace 1 that looks like Figure 14 has no indication that A → C does not pass the threshold, as it appears identical whether or not the reformatted trace was transitively reduced. For instance, if $A \rightarrow C$ did indeed pass the threshold and we transitively reduced the reformatted Trace 1, it would once again only leave $A \rightarrow B$ and $B \rightarrow C$, just as in Figure 14.

4.9 Our Assumptions

We discuss and list here the assumptions for accurate functioning of our algorithm.

We take information about each span to create a reference to it in the distributed system – its nameID. The creation of a span's nameID is detailed in Section 4.4. We assume that this naming method is enough to make all spans unique in a given trace (see Section 3.1 for an idea to restrict the scope of this assumption).

Zipkin [9] and Dapper [2] both use "multi-server" spans. This means that the client and the server in an RPC could share a spanID (they would be part of a single span) [6]. We assume that our traces use a "single-host" model, so each span belongs to only one service. This type of model is used by more modern tracing frameworks like Jaeger [10].

We also assume that the timestamps inputted in the trace data are global timestamps (i.e. they have been adjusted for clock skew, which Jaeger does do).

Finally, we share the one of the main assumption on which *The Mystery Machine* relies, as detailed in Section 3: all the trace data has enough variation to accurately determine the a representation of the true causal dependencies in the GCM.

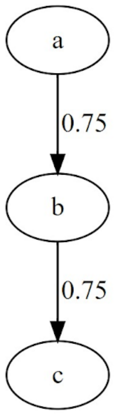

Figure 14. Reformatted Trace 1 with First Order Transitivity Failure

5 Future Work

5.1 Evaluation

We are currently in the process of evaluating our implementation of the algorithm. We will likely use test traces from DeathStarBench, which is representative of common distributed systems [8].

5.2 Addressing Repeated Events with Backtrace Information

We hope to partially address the problem of repeated events (discussed in 3.1) by implementing backtrace information for each span into our algorithm. The spans in the traces can include the parent information, therefore allowing us to determine the backtrace of each span. We could then distinguish spans that currently share nameIDs by their backtrace.

This potential method would mitigate the issue of repeated events at least somewhat by alleviating errors caused by repeated events under different parent spans by helping distinguish these spans as such.

However, what if the parent spans are distinct but have the same operationName and serviceName (these are the two primary components of our nameIDs as we currently construct them — this process detailed in Section 4.4)? Extending the backtrace farther back — that is, perhaps including the span's "grandparent" as part of its nameID — could alleviate errors caused by this. This would be an instance of extending the *backtrace depth* of a span.

Nevertheless, there is an accuracy trade-off. If we increase the backtrace depth too much, then the same span will not be recognized if those backtraces differ even slightly.

5.3 Hadoop Implementation

We are currently working on implementing our program on Hadoop [11], a distributed computing framework, in order to make our program scalable considering the large amount of data that distributed tracing generally produces for processing.

A Hadoop MapReduce framework consists of two primary components – a mapper (to perform tasks on input files) and a reducer (to simplify/process the outputs of the mappers).

In our implementation of Hadoop, a map task will be created to process each trace file (each trace is in its own file), and the traces will be stored in JSON format as shown in Section 4.4. Each map task will take the events in its trace and output all happens-before and simultaneity relationships it finds as per the timestamps. These relationships will be outputted as `<key,value>` pairs for the reducer. The key of each pair will be the relationship as an ordered `String` (e.g. `"A_S -> B_E"` for a happens-before relationship from event `A_S` to event `B_E` in a trace) while the value will be the number of times the relationship is observed in the trace. We use "->" to indicate happens-before relationships and "=" for simultaneity events. For simultaneity relationships, we also order the events on either side of the equals sign alphabetically (this is to prevent the map tasks from producing the two differing keys `"A_S = B_E"` and `"B_E = A_S"`, which express the same type of relationship).

The reducer will then sum up the values for each distinct key to get the total number of times each relationship occurred. From there, this data will post-processed to create the GCM based on the threshold condition.

Our Hadoop code thus far can be found at `https://github.com/TanmayGupta23/ScoobySystems`. Note that we chose to implement our MapReduce program in Java.

6 Acknowledgements

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

References

[1] 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`.

[2] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010. URL `https://research.google.com/archive/papers/dapper-2010-1.pdf`.

[3] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. So, you want to trace your distributed system? key design insights from years of practical experience. *Parallel Data Lab*, 2014.

[4] 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, April 2007. USENIX Association. URL `https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework`.

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

[6] Y. Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing, 2019. ISBN 9781788627597. URL `https://books.google.com/books?id=4AuLDwAAQBAJ`.

[7] Denes Vadasz. Open for event based tracing? *The Medium*, 2019. URL `https://medium.com/opentracing/open-for-event-based-tracing-a326c295f2a2`.

[8] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud amp; edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming*

Languages and Operating Systems, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi:10.1145/3297858.3304013. URL `https://doi.org/10.1145/3297858.3304013`.

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

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

[11] Apache hadoop, 2021. URL `https://hadoop.apache.org/`.