Distribution Agreement

In presenting this thesis as a partial fulfillment of the requirements for a degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis in whole or in part in all forms of media, now or hereafter now, including display on the World Wide Web. I understand that I may select some access restrictions as part of the online submission of this thesis. I retain all ownership rights to the copyright of the thesis. I also retain the right to use in future works (such as articles or books) all or part of this thesis.

Tao Zhou April 12, 2022

Automatic Anomaly Localization in Distributed System

By

Tao Zhou

Ýmir Vigfússon, Ph.D.

Advisor

Computer Science

Ýmir Vigfússon, Ph.D.

Advisor

Nosayba El-Sayed, Ph.D.

Committee Member

Avani Wildani, Ph.D.

Committee Member

2022

Automatic Anomaly Localization in Distributed System

By

Tao Zhou

Ýmir Vigfússon, Ph.D.

Advisor

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Computer Science

2022

Abstract

Automatic Anomaly Localization in Distributed System
By Tao Zhou

The complexity of a distributed system with large-scale applications poses a great challenge to diagnose its anomalous behaviors, such as faults, high latency, and others. Although many solutions have been proposed to analyze system anomalies, they all have their limitations. Acknowledging the difficulties to identify root causes of anomalies, we avoid the common approach that, through either statistics or inference, attempts to identify root causes. In our thesis, we designed and implemented a structure that can automatically detect and localize anomalies in a distributed system, with the help of retrospective sampling, our self-defined attributes, and a modified Association Algorithm. Our project performs real-time per-triggered-trace analysis and produces a prioritized list. Every entry is a set that contains information about the possible locations of root causes and the higher rank corresponds to a stronger association with the anomaly. When anomalous symptoms are detected in the system, the operator is expected to quickly locate the root cause with the help of the prioritized list.

Automatic Anomaly Localization in Distributed System

By

Tao Zhou

Ýmir Vigfússon, Ph.D.

Advisor

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Computer Science

2022

Acknowledgments

I would like to give special thanks to Prof. Ýmir Vigfússon, who advised my honors thesis and provided kind support and patient guidance. Without his encouragement, I would never finish this thesis. Neither would I feel confident to apply for the Ph.D. programs. I also want to thank Prof. Nosayba El-Sayed and Prof. Avani Wildani for lending their time and advice for this thesis. Moreover, I am grateful to Prof. Jonathan Mace and Dr. Lei Zhang, who provided great help with the project. Last but not least, I am thankful to everyone in the Simbiosys (which is definitely world's nicest group) for their kindness and help.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Today, many companies embrace the design of the distributed system. What used to be handled by one or several monolithic machines is now broken down into smaller microservices (or nodes) that provide fewer and more specialized functionalities.

The microservices architecture offers many benefits as compared to the monolithic one. And one of the most important advantages is fault tolerance. Previously, if the machine that holds a service is down for maintenance, all the services held on the same machine will be available; while the shutdown of service in a microservices architecture does not necessarily disable other services. For example, the shutdown of the login service may not prohibit a user from accessing other services, if the user has acquired a login token before. On the other hand, multiple nodes can handle the same type of services, managed by a load balancer, to ensure its stability.

It also has many other benefits, including a lower long-term cost, better scalability and modularity, and also efficiency in team development. However, the increasing complexity is definitely not one of them.

For a large-scale distributed system, which may contain hundreds of nodes, the network that connects these nodes forms a complex graph. The failures of machines are usually unpredictable; now with numerous nodes sitting in a network where ev-

ery node can potentially fail, we would expect to see more problems arising from a distributed system. Therefore, it is crucial to design some new techniques to address the growing problems effectively.

Diagnosing a complex system is undoubtedly difficult. Consider a request handled by multiple nodes. A wrong output at an earlier node causes the anomaly at a later node. How does an operator supposed to start from the place where the anomaly is detected and trace all the way back to the actual culprit, given that the nodes are only loosely connected through the local network? And within the numerous requests being processed, how can the operator find out which request contains abnormal symptoms? All these questions await practical and effective solutions.

The wisdom of the masses is inexhaustible; many new methods are proposed to answer these tough questions. They can be abbreviated in two steps. First, to keep track of requests, some important information is collected at local nodes and centralized. Then, state-of-the-art theories and algorithms provide the fundamental support for thorough analysis based on the collected information.

If the problems can be perfectly solved, this thesis will merely be a monotonous survey of the proposed solutions; instead, it turns out that many existing methods face great limitations.

Take the information analysis part for example. Machine learning is a very hot topic and undoubtedly numerous relevant techniques have been designed to determine the root causes when anomalous symptom arises. But machine learning requires the active training of existing data, which gives little guarantee about its effectiveness against unexpected failures. Other methods, although quite different from machine learning, also face their own limitations.

In fact, even the process of categorizing and analyzing all types of failures can be very difficult; otherwise "unexpected failure" would seldom become a frequent phrase in the system. Thus, we are not here to provide an unmatched solution that can

determine the root cause of anomalies once and for all and challenge the efforts of other researchers. Instead, we would like to rethink the problem from a different perspective and propose a less ideal but more practical solution. We make a small compromise to answer the question of "where is the root cause" in replace of a "what" question.

In this thesis, we present our design and implementation of architecture available for real-time anomaly localization in a distributed system.

Chapter 2

Background

A request throughout the system can be modeled as a **trace**, which contains information from all nodes the requests has visited. Figure 2.1 illustrates a simple request processed by four nodes, where the arrows between nodes represent the remote procedure calls. The trace consists of data generated at all four nodes and may also contain additional information such as the order in which nodes process the request.

Traces provide an efficient way to monitor requests throughout the system. And thus, the anomaly analysis, or the analysis of irregular, detectable symptoms in a system, involves the analysis based on trace data. When the operators see an anomaly occurring in the system, they will try to identify the root causes based on the clues given by the trace data. Due to the potentially huge amount of traces, this process is usually facilitated or even completely handled by analyzing programs.

In general, the entire diagnosing process can be split into two parts: trace collection and trace analysis.

2.1 Trace Collection

There are many open-source, well-designed, and widely-used trace collectors in the industry, including Canopy from Facebook [8], Dapper from Google [15], and many

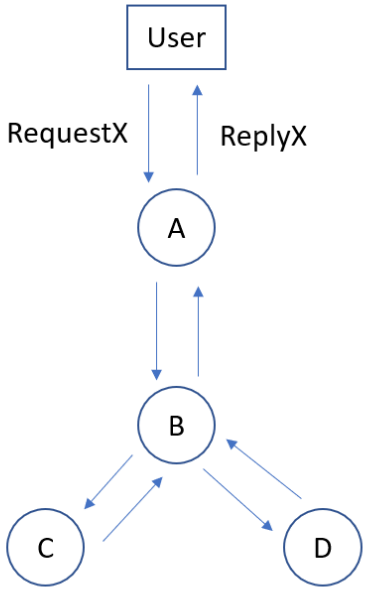

Figure 2.1: An example request.

more [2, 3, 4, 7]. Although with different emphasis and design choices, most of them follow a common structure, with agents deployed on every node to generate traces and a central place to collect trace data. Usually a trace will carry a unique identifier throughout the system, so that the collector can keep track of all its data and reconstruct the trace.

Trace data are generated at tracepoints, usually through the process of instrumentation of the source code. There are also other schemes such as pivot tracing, which allows dynamic instrumentation [12].

Agents usually hold passive roles in the trace collection process. To facilitate the collection of trace data, it may occupy memory for temporary data storage, and send data back to the collector in larger data trunks, through the remote procedure calls. They function like worker bees, following the rules set by the collector. For example, agents usually do not make decisions about whether collecting or dropping a trace; they seldom process the collected trace data. As we shall see later, the data collector

we build our project upon, Hindsight [18], relieves the burden of central collector and assigns more active roles to the agents.

Most existing trace collectors provide full support of trace aggregation and collection. They provide ways to visualize requests as traces and also the underlying system as a graph. However, many of them implemented none or only rudimentary methods for trace analysis; instead, they provide API for customizing the analysis functions.

2.2 Trace Analysis

In this section, we briefly mention how other researchers use traces to analyze system anomalies and their respective limitations.

2.2.1 Machine Learning

Many systems use machine learning to perform statistical diagnoses [1, 9, 5]. They heavily rely on either the normal traces or traces labeled with anomalous symptoms for data training and then use the learned models to make the subsequent diagnosis. It has several limitations due to the nature of machine learning.

1. Machine learning might be a useful tool to detect known failures. But when a new and rare anomaly occurs in the system, its effectiveness is not guaranteed. In general, we would expect a responsive system toward all types of anomalies, instead of merely the familiar ones.

2. Because of the complex design of distributed systems. The model trained on one system may not work as well on another.

3. In practice, healthy traces outnumber abnormal traces. Many medium or small-sized distributed systems may not generate enough abnormal traces for training.

2.2.2 Trace Tree & Provenance

Several systems use trace trees [15] and provenance [16, 6, 19] to perform diagnosis. Although two different methods, both function in a similar way. They acquire details about a trace's shape and structure and make inferences based on the known information. For example, Dapper treats a trace as a tree of spans where every span represents a node or a service. These spans are linked by span IDs so that when the trace data are collected at a later point, the complete trace tree can be reconstructed. However, the method has its own limitations:

1. A data center may have millions of events happening in the system, many of which could be relevant to the symptoms [17]. The complexity rockets rapidly as the number of traces the system needs to investigate increases. Moreover, the many possibilities of anomalies make it hard to predict accurately.

2. In practice, the symptoms of an anomaly may be distant from its actual location. The inference will be difficult in this case, because the process usually involves a reasoning beginning at the point where the abnormal symptom is detected.

3. Because trace trees and provenance emphasize more on the order of which nodes process the requests and make inferences based on the information provided in a specific trace, it does not fully investigate similarities and differences between different traces. For example, a cross-trace analysis may quickly discover that all requests go through Machine X are abnormal while those processed by Machine Y shows no symptom. However, it may take more efforts to track from the symptom back to Machine X using trace trees or provenance.

2.2.3 Zeno

To address the second limitation of trace tree and provenance, Zeno introduced the Temporal Provenance [17], which pays more attention to the sequence of events in

which the requests are processed at local nodes. It also considers relevant traces that visited the same node. However, Zeno is restricted to time-related issues only.

2.3 Our Goal

We rephrase our goal in more detail. We aim to design a structure that can automatically detect and localize anomalies occurred in a distributed system, with the following properties:

2.3.1 Anomaly Localization

To find the root cause of an anomaly, or the true reason behind causing the anomaly, is seldom easy. An abnormal symptom such as the unusual delay of a response may be explained by a unstable network, too many requests at some nodes, or even bugs in some code that prolong the processing speed. In fact, the procedure of root cause analysis involves eliminating the unfitting explanations and report the more likely ones.

Because there are so many possibilities of causes of failures and anomalies in a system, it is already difficult enough to categorize them, not to mention producing a practical analysis for each of them. Moreover, root cause analysis may not show the result accurately in case when an unexpected error occurs. Due to its difficulty, many existing analyzers instead only focus on a subset of possible causes [11, 10, 17], whose functionalities are limited especially in an industrial setting.

We take a different approach. We do not make direct root cause analysis in the system. Instead, we shift our focus from diagnosing the problem to providing anomaly localization. That is, we only output a "treasure map" that points out the possible locations of root cause, while the actual work of determining the root cause is left to the operators and developers.

Notice that the "where is the root cause" problem is more like a compromise of the "what is the root cause" problem. Given better granularity of trace data, our solution should be able to give stronger indications of the true root cause. For example, suppose some problem arises in the system due to a wrongly used variable in the codes on some node. If the traces only provide general information of the nodes, our program should be able to locate the problem back to that node; however, if more information about the codes are collected, our program may locate the problem to the codes or even the variable directly. However, it is the developer's job to figure out why the variable causes the problem.

Moreover, anomaly localization approaches the problem in a way agnostic of the underlying cause, which allows our program to tackle a wider range of system anomalies. It also allows us to make analysis regardless of the underlying structure of the distributed system.

2.3.2 Trace Filtering

Unlike many trace collectors which either collect traces unanimously or only provide simple sampling schemes [15, 8, 2, 3, 18], we expect our program to effectively filter out irrelevant traces before performing analysis. The sampling we will use is called retrospective sampling. The next section is devoted to explanations and implementations of this sampling scheme.

2.3.3 Trace Aggregation

As mentioned earlier, many traditional methods do not take full advantage of the information across traces. Our program should provide a mechanism for aggregating traces and making cross-trace comparisons.

2.3.4 Portability

The flexibility of the structure of distributed system requires a general analyzing mechanism that does not rely on its structure. Many existing methods, especially those involving machine learning, require the active training of data in an actual system. The resulting program, however, may not work as well in other systems due to the complexity of their structures. To make the program portable, we do not infer based on the underlying structures of the system and traces; we rely on the trace data only.

2.3.5 Responsiveness

Whenever an anomaly occurs in the system, the operators and developers need to identify and solve the problem as soon as possible, to give minimal influences on the existing and incoming user requests. Some serious problems, such as bugs exploited by malicious attackers, may put the entire system on risk if no timely actions are made. Therefore, it is very important to provide analysis results immediately after an abnormal symptom is detected. More specifically, our we expect our program to perform real-time analysis for every trace that shows some abnormal symptoms and output the possible locations of root causes within seconds.

2.3.6 Visualization

The output should be presented in a visualized way to facilitate the subsequent system diagnosing. A per-trace prioritized list will be printed out, showing operators different emphasis on several possible localization results.

Chapter 3

Motivation: Hindsight

3.1 Head-Based Sampling

In practice, healthy traces outnumber abnormal traces. In companies such as Google, which can potentially collect millions of traces every day, only a small portion of them will explicitly show symptoms of anomaly [8]. This is the demonstration of a well-constructed distributed system, but also increases the workload of analyzing irregular symptoms due to the unbalanced data size.

The truth is that not all traces are relevant to ongoing anomalies. The nature of distributed system breaks the burden of request handling into smaller pieces of work handled independently by nodes loosely connected through a local network [14]. What usually happens in a large system is that the machine which experiences a temporary failure or slowdown only processes a portion of the requests, and many traces do not provide information about the system.

For instance, traces that represent login requests processed at one node will unlikely contain information about what causes the unexpected outputs of the multiplication operation processed by another node (assume it does not verify the login token). Only these traces that record the input and output of the similar operations

will be helpful for investigation.

In practice, we only want to investigate traces showing the symptoms and relevant traces (for instance requests processed by a different machine with the same copy of codes), and therefore a sampling scheme to sift out less useful traces are desired. Moreover, sampling also helps reduce the data size to guarantee memory and time-efficient analysis.

What we see from the popular data collectors is that most of them only provide a partial or complete implementation of head-based sampling [8, 15, 3, 2, 7, 4]. In another word, when we use these collectors, the decision to keep or drop a trace must be made before the request is processed in the system. This is most evident in Zipkin's before-the-fact sampling [3]. While others claim to provide more advanced sampling schemes, such as Dapper's adaptive sampling [15], its nature still prohibits the delay of decision making.

We would like to use in our program a tail-based sampling. That is, we want to delay the collection of traces to a later point when the request has been partially processed in the system and we have acquired more evidence about its well-being.

However, traditional tail-based sampling methods have great limitations in case when there is an enormous amount of trace data generated. It imposes a great workload to the backend, which still needs to make independent collection decisions for every trace, and sometimes the head-based sampling is necessary to mitigate overheads. To address this, we borrow the idea of retrospective sampling from Hindsight.

3.2 Retrospective Sampling

Hindsight proposed and implemented its tail-based sampling method, retrospective sampling [18]. Unlike traditional tail-based sampling methods, Hindsight by default do not collect any trace data. Every nodes, however, still generate all the trace

data, but store them temporarily and locally. A signal is then required for the active collection of a specified trace.

Because our project is built upon Hindsight to take full advantage of retrospective sampling, it is necessary to elaborate on the details of its underlying architecture. Later sections will cover our implementation procedures.

3.2.1 Basic Architecture

Its overall architecture is still very similar to Dapper, where node-based agents are responsible for trace data generation and a central collector can reconstruct the complete trace by requesting data from agents. Unlike Dapper, agents in Hindsight need to maintain local and temporary storage for trace data using the node's memory. Traces are collected in a selective manner: a "trigger" signal, issued either automatically by any agent or manually by the central collector, begins the transmission of trace data from agents to the collector.

3.2.2 Data Generation

Hindsight allows generating trace data by calling its `tracepoint_tracepoint` function. Instrumentation of the source code is required to generate trace data. Hindsight does not define any specific data structures; the input to this function is simply a byte sequence of arbitrary length. The input will then be copied into the temporary storage.

3.2.3 Data Storage

Hindsight uses a fixed-size queue of trunks to store trace data, every trunk has an equal size, typically several megabytes, defined during the initialization process. The agent only assigns one data trunk to every trace at the beginning of its life; every

subsequent data generated at any local tracepoint about this trace will be written into its assigned data trunk. To guarantee the integrity of a trace, a trunk can only be written by one trace at any time.

A trace can potentially have a large data size. The agent will recursively assign a new data trunk to the trace if its previous one is full. From a broader view, the agent maintains a map from every recorded trace to its corresponding data trunks.

When all data trunks are occupied, Hindsight regularly evicts the older traces and reallocates the trunks to new traces. Therefore, every trace, unless triggered, will have a limited timespan.

3.2.4 Trigger Mechanism

A trigger signal is required for the collection of trace data. Specifically, Hindsight provides a trigger function for agents, which can be automatically called through instrumentation. The local agent will henceforth deliver this signal to the central collector. Alternatively, the central collector can manually set up a trigger if it knows the trace IDs.

3.2.5 Trace Collection

The central collector keeps track of the trace using breadcrumbs. Every breadcrumb takes the address of either the last node visited or the next one. Therefore, by recursively acquiring the breadcrumbs, starting at one known agent, the central collector can reconstruct the complete trace "tree." In fact, the actual structure of trace matters little to our project, but we use these breadcrumbs implicitly to acquire all trace data.

3.2.6 Limitations

While the retrospective sampling allows us to sift out irrelevant traces, Hindsight is far from the ideal architecture to address our goal.

The first issue is that, because a trace's data are maintained by agents independently and only temporarily, the timespan of a complete trace is the minimum time it stays in any agent. While partial traces still may contain useful information, they have limited value in diagnosis [7, 15]. Therefore, it is desirable to find methods to extend the time span of a triggered trace. The rough idea is to reduce the data size by prohibiting verbose logs and restricting data structures. This gives rise to our design of the attributes, elaborated in the next section.

The second problem is that Hindsight does not implement any scheme to effectively group traces. The central collector only knows where the data of a trace come from, but it does not keep track of which agent is responsible for the generation of which part of the data. We will also see soon how our self-defined attributes address this problem through trace aggregation and comparison.

Chapter 4

Design

This section covers the definition of and reasoning behind our self-designed attributes, and how to use it in a mathematical way to investigate the problems in a distributed system.

4.1 Attribute

We define an **attribute** as a 2-tuple: {*key, value*}. It is the fundamental unit of trace data in our design and is also the only data structure involved in trace generation and aggregation. Broadly speaking, both key and value are information revealing the details of a distributed system, but they have vastly different emphasis.

An **attribute key** is a string that represents the information of tracepoint, or more specifically, what will be collected uniformly from every trace. Examples include *MachineID*, *ServiceType*, and *OutputValue*. It is a constant tag bind to a tracepoint, assigned during the process of instrumentation. Every subsequent trace that visits this tracepoint will be assigned an attribute with this constant key.

An **attribute value** can be either a string or a Boolean. It represents the trace specific information with respect to the key. For instance, a key *ServiceType* may assign values *Business* or *Personal* to different traces, based on the details of that

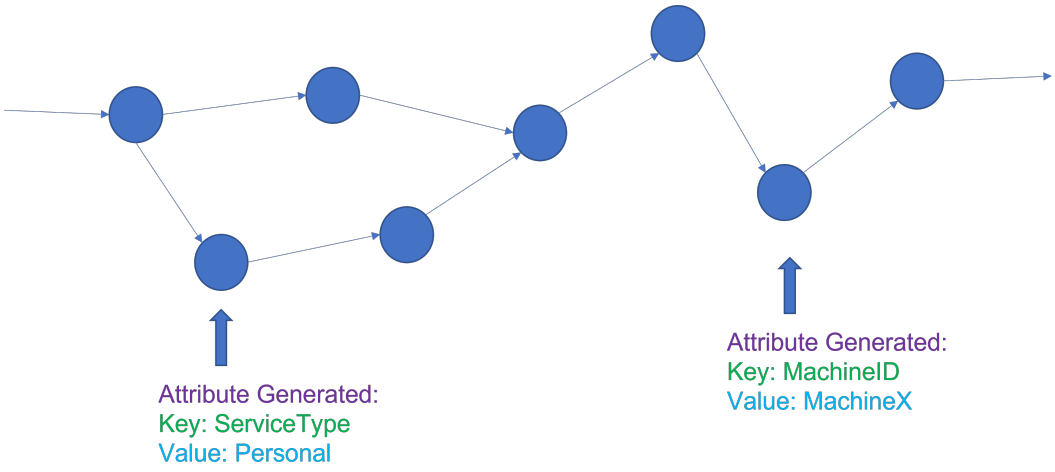

Figure 4.1: An example trace with two attributes.

trace. It can also assign a constant value string to every trace, for example a key *MachineID* may assign value *MachineX* to every trace, because the traces are processed by machine X.

Figure 4.1 gives an example of a trace with attributes. Every circle represents a node and the arrows tell the flow of the request throughout the system. Two attributes are generated at two different nodes: {*ServiceType: Personal*}, {*MachineID: MachineX*}.

Different tracepoints may assign the same attribute key to traces. Continue from the *MachineID* example above, two machines running the same services managed by a load balancer will assign the same key *MachineID* to every trace but different values, *MachineX* or *MachineY*, based on which machine handles the request.

There is only one special attribute not assigned by any tracepoints, which we call the **anomaly attribute**. It has the constant key string *anomaly* and a Boolean value indicating whether some irregular behaviors (unexpected output, delayed response, etc.) of a trace has been detected by either the automatic trigger system or an operator manually.

4.2 Attributes Enable Trace Aggregation

The sole purpose of attributes is to provide an efficient way to collect and aggregate trace data, while allowing fast and straightforward comparisons across traces.

As mentioned before, there are so many possibilities of failures or anomalies in a system that there is hardly a way to treat every type equally and provide a general solution to the problem. However, it does not mean we can have no progress when a new type of failure comes in. The design of attributes helps change the focus from determining the actual failure or anomaly to instead providing a practical solution of localizing the problem. Consider this in a specific example.

Example Scenario. Suppose the same requests go through a load balancer, which assigns two different machines X and Y to process them. It turns out that machine X and Y produce different outputs even for the same requests and those requests handled by machine X are detected to have abnormal symptoms.

What does an operator usually do? Immediately investigate machine X to find out if its output is irregular and therefore causes the anomaly. The truth is, diagnosing a complex distributed system is a demanding work and even finding out "there are output differences between X and Y" will take much effort.

Attributes allow the fast detection of irregular patterns. By defining attributes indicating the machine ID that process the request for every relevant trace, we can quickly compare different traces and find out that traces labeled with attribute {*MachineID: MachineX*} has a much higher anomaly rate than those with {*MachineID: MachineY*}. This allows us to mark Machine X as an interesting place to investigate.

Besides, the granularity of localization depends on the choices of attributes. Generally speaking, more attributes collected from a distribute system will give us a more detailed image of the conditions and behaviors of the system. An attribute showing

that Machine X uses an older version of the code than Machine Y allows us to further locate the problem to the code on the machine.

Moreover, comparisons need not be made on one attribute only. Instead, we should think about the localization problem as producing an attribute set. These attributes together give operators indications about where the problem most likely occurs. Due to its nature, we require the trace to contain every attribute in the set. In another word, the resultant attribute set is a subset of all attributes of the analyzed trace, as illustrated by the following example.

Example. Suppose traces that go through machine X and perform the multiplication operation have an unusually high abnormal rate. We can produce the comment above through mathematical investigations of the behaviors of traces labeled with the attribute set {*MachineID: MachineX, Operation: Multiplication*}, as compared to other traces.

It is also important to know which traces should be compared to, rather than taking all traces into consideration. This will be covered in a later section.

To account for the errors and deviations in producing such a set, we will instead provide a prioritized list, in which the higher rank represents a stronger association between the attribute sets and the anomaly. Mathematical tools are desired to generate such a list. We will discuss our motivations from the Association Rule first.

4.3 Motivation: Association Rule Mining

Ideally, we expect to define a "level" of connection between a set of attributes and the existing abnormal symptom. A similar method exists in the market basket analysis, named **Association Rule Mining** [13].

For Supermarkets and on-line shops, learning the common shopping behaviors of

their customers can be very helpful for increasing sales. For example, if two products are almost purchased together, a store can deliberately place them distantly to ensure a greater exposure of other products to the customers. It can also produce targeted advertisements based on customers' existing shopping behaviors, by suggesting items purchased by other customers with a similar shopping pattern.

To investigate the strength of association between an existing set of products and another product, the Association Rule Mining models every transaction as a set, where the entries are the products in one purchase.

The "Beer and Diapers" Example. Although it sounds quite counter-intuitive, researchers find that people buying diapers are more likely to purchase beer [13]. The problem of determining whether there is a connection between beer and diapers can be model as finding the association between the set with only one entry, "beer", and the item "diapers".

4.4 Interest of Association

The Association Rule defines an **interest** to reflect the strength of association. Roughly speaking, given a nonempty set X and item Y, it calculates how much does the existence of X increase the likelihood of Y. Taking the beer and diapers example, the association between beer and diapers is quantified by the difference of likelihood to purchase diapers between a customer who purchased beer and an average customer. To put it more formally:

Definition. Suppose every transaction is model as a set of purchased items, then the interest of association between a nonempty set X and item Y ($Y \notin X$), written

as $\{X\} \rightarrow Y$, is defined as

$$\frac{\#\text{ sets containing } X \text{ and } Y}{\#\text{ sets containing } X} - \frac{\#\text{ sets containing } Y}{\#\text{ sets}}$$

A bigger interest generally represents a stronger association; the interest can also be negative, which means a very unlikely association. To determine which set X promotes item Y most, it is necessary to calculate the interest of $\{X\} \rightarrow Y$ for all possible X.

4.5 Interest of Attribute Sets

The reason we mention the association rule is that the method can be used similarly for attributes. If we represent every trace as a set which contains all its attributes as set elements, we can turn the question "which attributes of this trace are more relevant to the anomaly" into "how strongly are the attribute subsets of this trace associated with the anomaly attribute".

A similar interest formula can be defined for an attribute subset X (for convenience, we use Y to represent the anomaly attribute):

$$\text{interest of } \{X\} \rightarrow Y = \frac{\#\text{ abnormal sets containing X}}{\#\text{ sets containing X}} - \frac{\#\text{ abnormal sets}}{\#\text{ sets}}$$

However, this formula does not work as desired. The second term (after the minus sign) is a constant for any attribute subset X! And the first part merely tells the abnormal rate among all traces sharing attributes in X. The example below describes a scenario where the formula produces unwanted result.

Example Scenario. For convenience we ignore the second constant term. Suppose a user U sends the same number of requests to machine A and B, where all the requests

handled by machine B exhibits abnormal symptoms while those go through machine C are fine. Intuition tells that the problem might be associated with machine A, and the user U should be as innocent as machine B. But calculations show that interest of $\{B\} \rightarrow Y = 0$ while $\{U\} \rightarrow Y = 50\%$. They have different interests!

The reason behind this phenomenon is that we are making improper comparisons between different types of attributes. More specifically, instead of asking "is machine X associated with the anomaly", we should ask "compared to other machines, is machine X more associated with the anomaly". The comparisons should be drawn between attributes that share the same key and only the extent to which an attribute outperforms its similar attributes (that is, attributes with the same key) should be compared across different types of attributes. With this in mind, we introduce the modified Interest of Attribute Sets.

4.6 Modified Interest of Attribute Sets

We first define a similar attribute set to a set X to be a set that shares the same attribute keys with X, but possibly different attribute values. Then, the general question we would like to ask is "compared to other similar attribute sets, to what extent does the presence of an attribute set increase the likelihood of anomaly."

This is quantified by the modified interest of an attribute set:

$$\text{interest of } \{X\} \rightarrow Y = \frac{\#\text{ abnormal sets containing X}}{\#\text{ sets containing X}} - \frac{\#\text{ abnormal sets containing keys of X}}{\#\text{ sets containing keys of X}}$$

Given a triggered trace, this is the general formula used to calculate the strength of associations between its attribute subsets and anomaly. A prioritized list of its subsets will then be generated by ranking the results derived from the formula above.

The following section will cover the underlying algorithm to implement the prioritized list.

Chapter 5

Implementation

In this section, we go through the details of the underlying architecture we choose, from the generation, temporary storage, and collection of attributes to the calculation procedures and derivations of the prioritized list. Our current implementation is built upon the existing Hindsight project for its full support of retrospective sampling. For convenience, we also modify the other parts of its code to reduce our coding workload.

However, this is not to say that the implementation must be bound to Hindsight. Neither are the attribute manipulation procedures mandatory. Other implementation choices may achieve the same effect with a differently designed attributes database and a distinct algorithm to calculate the interests of associations. We loosely require the other ways of implementation to contain the following components:

1. Attribute generation and temporary storage,
2. Attribute filtering using retrospective sampling,
3. Per-triggered-trace attribute collection of all relevant traces,
4. Prioritized list generation using the modified interests of associations.

5.1 Attribute Generation

Hindsight provides a great implementation of tracepoint instrumentation, but with undefined trace data structures. All data are instead considered as a sequence of bytes, which are then grouped by fixed-size trunks. To fully take advantage of the instrumentation methods, while allowing the process of our self-defined attributes, a new wrap function is designed for writing attributes. It reads in and encodes an attribute and calls the regular Hindsight function to write encoded attributes to its data trunks. A decoding function is called subsequently to retrieve the attribute from the data trunk and store it in a separate attribute database.

5.2 Attribute Database of Agent

Every agent maintains a lightweight database for temporarily storing attributes. It keeps a hash map of all the attribute keys generated at tracepoints within its duty. Every key object also contains a hash map for its associated attribute values. To keep track of all the traces that generate the same attribute, we use a list structure for every value object. Moreover, to retrieve attributes for one trace fast, another list is used by every trace to record all its attributes. Be aware that every agent handles its data independently. Hence it is impossible for a database located at one node to contain attributes generated at another node. Figure 5.1 simplifies and visualizes the underlying database structure. Broadly speaking, the database mimics a doubly linked structure, where attribute keys, values, and traces are treated as distinct objects, and every line in the graph connecting two objects represents a two-directional link. The nature of this structure guarantees fast query operations for both attributes and traces.

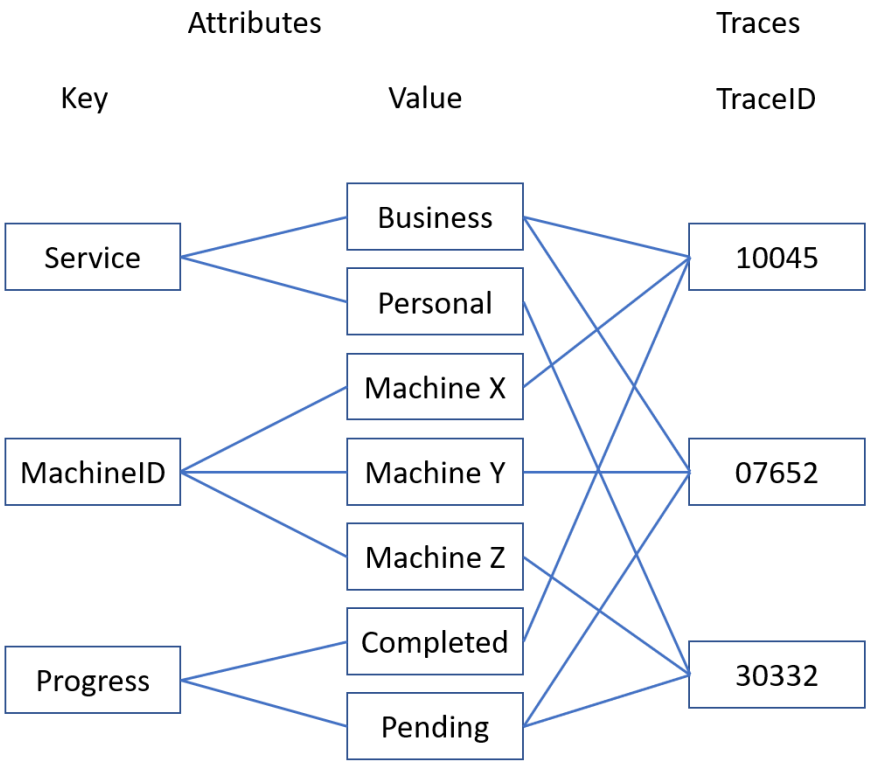

Figure 5.1: A doubly linked structure of attribute database.

5.3 Database Functions

We developed five functions for the attribute database, listed in Table 5.1.

Table 5.1: Interface for database manipulation.

Function Name	Input	Output
`AddAttr`	Attribute & Trace ID	
`RemoveAttrs`	Trace ID	
`FindTracesByKey`	Attribute Key	List of Lists of Trace IDs
`FindTracesByVal`	Attribute	List of Trace IDs
`FindAttrsByTrace`	Trace ID	List of Attributes

The agent calls the `AddAttr` function automatically to write a newly generated attribute into the database. The process includes creating the new attribute key, value, and trace objects if they are nonexistent and updating links between these objects. This is the most time-costly function and the speed relies heavily on the efficiency of searching and updating the list structures. It is suggested during the instrumentation process to generate attributes with keys mapped to only a small number of possible values. In another word, attribute keys such as "user" and "IP" are not recommended when there are potentially too many different usernames or IP addresses. This is not only beneficial in making our database more responsive to attributes when the number of requests surges, but also helpful for later analysis to reduce the effect when the sample size of traces under an attribute is significantly outnumbered by the sample size under the same attribute key but allows other values.

Older traces are less helpful than newer traces in providing a recent view of the system, therefore we need to set up a timeout for every trace and regularly delete the outdated traces. We rely on the built-in timeout mechanism of Hindsight, which ranks traces based on the last time it generated data and removes inactive traces to maintain a fixed maximum total data size [18]. Because every attribute occupies only a small amount of data, the database maintains an almost-constant number of traces at any time. It is possible that the size may drop slightly when several traces have

too many attributes generated at one node. To delete attributes from the database, a `RemoveAttrs` function is regularly called. Given a target trace ID, it removes the links between the trace and its relevant attributes, deletes the trace object, and also the attribute value object if it is not used by any other traces. On the other hand, the attribute key object is never deleted even if no values associated with it are present in the database.

Three additional query functions are built for the central collector to retrieve data from agents. They are evident by their name: `FindTracesByKey` allows querying for the ID of all traces containing a specific attribute key. The output is a list of lists in which IDs are grouped by the attribute values. Every trace ID is also accompanied by a Boolean value representing whether it was triggered before by the local agent. `FindTracesByVal` works similarly, only that the input is a complete attribute rather than merely a key. The third function `FindAttrsByTrace` performs a query in a "reversed" direction. After receiving a trace ID, it outputs all the associated attributes. All the queries are done through the gRPC communication between agents and the collector.

5.4 Attribute Query by Central Collector

The central collector collects attributes with the help of Hindsight's underlying structure. Suppose a local agent issues a trigger about a trace with ID "X" (for convenience we call it trace X). When the collector receives this trigger, it first uses Hindsight's breadcrumb mechanism to find out all the agents that contain information about trace X. However, unlike Hindsight, we pay no attention to the actual graph shape of this trace; we use the breadcrumbs only to guarantee that the central collector will visit every node that processed the request with respect to trace X.

The collector then sends `FindAttrsByTrace` queries to every agent discovered

through the breadcrumbs, to collect all attributes of trace X. To prepare for the subsequent analysis, it further sends `FindTracesByKey` queries once per attribute to agents where the attributes reside. If an agent does not generate any attributes about trace X, it is simply ignored.

To sum up, for a triggered trace X, we use several queries to collect its attributes and the ID of all traces which share at least one attribute key with trace X. The process starts when the central collector receives the trigger signal or manually issues a trigger (this happens when an operator believes a trace is abnormal nonetheless not signaled by any agents). The collector also keeps track of the recent trigger signal so that the request with respect to one trigger trace will not be performed twice.

5.5 Attribute Database of Central Collector

The central collector also maintains a database to store attributes received from queries. Overall, the database has the same structure as the ones implemented at local agents, only with several differences:

1. The attribute database at the central collector will be larger, as it may receive attributes through queries from any agent. Unlike local agents, it is desirable for the collector to hold more attributes and thus perform a thorough analysis upon larger data size.

2. Every trace is assigned an additional attribute with the key being the constant string “anomaly” and a Boolean value indicating whether the trace has been triggered before. A default value of False will be automatically assigned unless there are indications of trigger either through queries from the agents or manually triggered by the central collector. Databases at agents do not need this attribute because Hindsight records the trigger information locally for every trace.

3. Although not yet implemented, the database at the central collector will have its own independent timeout scheme. Recall that we rely on Hindsight's timeout when we implement the database at agents. However, Hindsight does not have a similar mechanism on the collector side. A self-built timeout mechanism is hence necessary to keep the size of the database at the central collector manageable. It is also beneficial to allow the customization of timeout so that it is not only based on the number of traces, as the agent-side database does, but can also work for other metrics such as the time since the trace was first introduced to the database.

5.6 Implementing the Interest of Association

After the required attributes for a triggered trace are collected, it is necessary to calculate the interests of attribute subsets mentioned earlier to produce the prioritized list. For any subset X, it is sufficient for us to find:

1. number of traces having all attributes in X,
2. number of abnormal (triggered) traces having all attributes in X,
3. number of traces having all attribute keys in X,
4. number of abnormal (triggered) traces having all attribute keys in X.

The interest of X can then be calculated using the formula provided by the modified association rule in O(1) time.

We start with only the single-item subsets. Because every attribute key has a map that records values, and every value object contains a list of associated trace IDs, these four numbers can be derived straightforwardly by counting numbers from the corresponding lists.

Interest for a subset with more than one element can be calculated through an inductive process. Suppose for every (k-1)-element attribute subset X of the given triggered trace ($k \geq 2$), we have built a list including all traces such that their attribute sets are supersets of X (this means that every trace shares all attributes of X). We then construct a similar list for every k-element attribute subset Y, by taking the intersection of lists from some (k-1)-element subset A and a single-item subset B such that:

1. attributes of Y = attributes of A $\cup$ attribute of B
2. list of Y = list of A $\cap$ list of B

It is then straightforward to count the number of (abnormal) traces having all attributes in any k-element subset of the given triggered trace. To count the numbers without considering the attribute values, repeat the process above but generate the list containing all traces sharing the same attribute keys.

Our derivations of interests follow a "layered" order: interests of k-element subsets will only be calculated after all interests of (k-1)-element subsets are derived. After we build the lists for all k-element subsets, all previous lists are no longer needed and hence destroyed. This layered structure is very useful. When the memory of the central collector is limited or a triggered trace has too many attributes, it is simple to restrict the calculations to only subsets with fewer elements.

Finally, we construct and output a prioritized list by ranking the interests among all subsets. The prioritized list will give an operator information about where should be investigated first when the anomaly happens in the distributed system, based on the locations that produce the corresponding attributes. It is worth mentioning that the entire analysis process is trace specific; The central collector will make the same calculation steps for every triggered trace, based on the order of trigger time.

Chapter 6

Evaluations

6.1 Sample Output

The example in Figure 6.1 illustrates a sample output of the prioritized list, where the traces going through "Node X" and "Machine A" are manually set at a 75% error rate (we manually trigger one of every four such traces). Only the single-element attribute subsets are calculated. Every entry consists of the elements of the set and a calculated interest shown in percentage.

Although the largest output 70.5% is very close to the 75% error rate, they nonetheless have different meanings. The numbers printed for each attribute subset only represent the relative strength of association between the subset and anomaly.

6.2 Experimentation

Because this research project is still in progress, we have only limited experiments. We expect to construct an environment that imitates an industrial setting, where its effectiveness can be thoroughly tested.

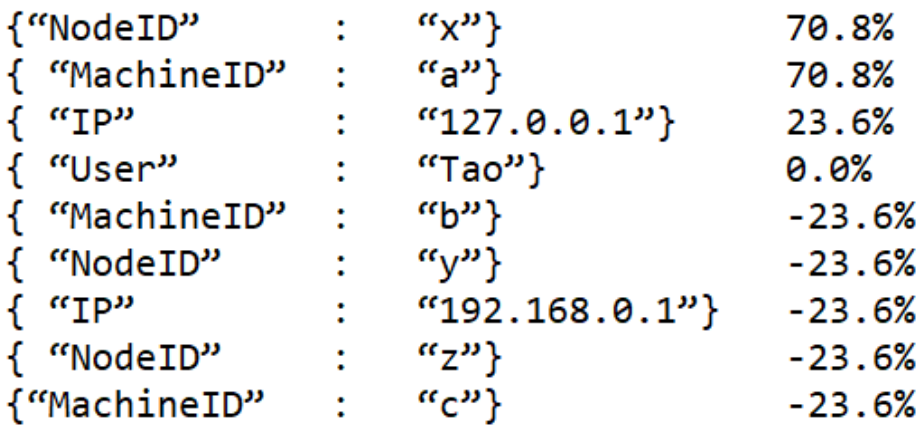

Figure 6.1: A sample prioritized list.

6.3 Theoretical Analysis of Efficiency

We use the Hindsight framework to generate attributes, the additional encoding and decoding functions are $O(n)$ in both time and space complexity, where n represents the total bytes of an attribute.

Data storage may be less efficient compared to Hindsight's built-in queue structure, especially when we use maps to store attribute keys and values in every database. However, in an actual setting, we expect the number of keys and values to be small for a more effective comparison between different attribute values. That means attribute keys such as username and IP address, which can potentially assign many different values to traces, are not recommended. In practice, we should see many traces share one or more attributes in the database. Therefore, the size of the database should be manageable. On the other hand, insertion and deletion take at most $O(N)$ time and query at most $O(MN)$ time, where M is the maximum number of values associated with one attribute key and N is the size of the largest list associated with attribute values.

For prioritized list generation, the time is proportional to the number of subsets. For each subset, a list intersection operation is performed whose time is linear with respect to the list size. As for the space complexity, we require $\binom{n}{k}$ lists maintained

for all k-element subsets at any moment. In practice, we expect to see the list size drop significantly after several intersection operations. Additional measures such as setting an upper bound for the element number of subsets may be applied to keep the time and space manageable, with the compromises of granularity of analysis.

Chapter 7

Limitations & Future Directions

7.1 Compatibility with Hindsight

Our project is implemented upon the basic framework of Hindsight to take full advantage of retrospective sampling and its local timeout mechanism for traces. However, the ideas behind the project are not dependent on Hindsight. In fact, borrowing the existing Hindsight framework significantly reduces the coding workload.

However, our current implementation does not work perfectly with Hindsight. For example, our attributes are stored twice locally, once in our self-defined attribute database and another time in the queue structure of Hindsight's agent in the encoded form. We will continue working to solve the compatibility issues either by modifying the underlying Hindsight code or building our own data generation and collection functions.

7.2 Existence of Multiple Anomalies

The occurrence of another anomaly will affect the prioritized list for an existing anomaly by reducing the calculated strength of association of unrelated attribute sets. If two triggered traces, each associated with a different anomaly, the order on

the prioritized for one trace is unaffected by the other, although the strength now is closer to 0. However, if two anomalies are closely related, the result might be quite different.

Consider an extreme case: suppose Machine A and Machine B process the same number of requests, each producing an abnormal output for every request. If *MachineID* is the only existing attribute key in the system, the result will only show 0 interest for both machines. Thus, new variables should be introduced to our original formula to ameliorate such an effect.

The example above tells that there exist cases where the association algorithm alone is less effective or even insufficient to address the problem. Therefore, we should put effort into improving the algorithms in the future.

7.3 Compatibility with Other Algorithms

Future efforts should be directed toward making our project more available to other possible algorithms. As mentioned above, the association algorithm we designed may be less effective in some cases, and due to the complexity of distributed systems, we would expect to see an algorithm working better in one system than another. Therefore, it is necessary for us to set the association algorithm as a default method while providing API for the implementation of new algorithms.

Chapter 8

Contributions & Takeaways

My contributions to this project include the design of the attributes and the modified Association Algorithm and the implementation of the following components:

1. the wrap function to generate and collect attributes upon Hindsight's raw data structure,
2. the doubly linked database to store attributes at the agents and the central collector,
3. the insertion, deletion, and query functions for storing and collecting attributes,
4. communications of query requests through gRPC,
5. the modified Association Algorithm and the prioritized list.

8.1 Beginning: Design of Attributes

Our research project started with the central idea of identifying root causes based on comparisons between traces, built upon Hindsight.

The key-value pair is nothing new in computer science, it is the basic element of many structures including maps. It is also widely used in data exchange formats such

as JSON. The pair itself is a simple but very efficient way of storing data, and when we can assign similar pairs to different traces, it helps us aggregate and finds similarities between traces. Thus, defining attributes was actually quite straightforward, as we needed a method to aggregate traces for ease of comparison.

However, the way we choose to collect attributes was quite different at the beginning. We believed that time is a very important metric for making root cause inferences. And therefore we were very passionate about collecting time-related data throughout the system, such as the processing time and the time spent in RPC calls. It turns out that time, recorded as a decimal number, is not easy to aggregate. Earlier designs include converting it to "categorical data" by keeping a record of the most recent 100 time values, and output only their percentile as the attribute values, instead of the actual time values.

Therefore, unlike our current version where attributes are defined loosely and only encouraged to have a small set of values (with minimal guarantee of aggregation), the previous versions give different standards for collecting and storing different types of data. For instance, we considered binary search tree and heap to update the most recent records of time and make a quick check of the percentile of time.

We even started categorizing errors into two large categories. Internal errors include problems that are not caused by user inputs, such as hardware failures and unstable network connections. This type of error usually has symptoms close to the root cause (very likely in the same place) and is easily detectable. External errors are those caused either by typical inputs or the way data is processed (for example, queue delay caused by too many recent requests).

8.2 Change of Mind: Anomaly Localization

Our categorizing efforts were not very fruitful. As mentioned earlier, there are simply too many possibilities of root cause given a single anomaly, and we had great difficulties in categorizing them, not to mention diagnosing every one of them effectively. We took a compromise: in order to detect all types of anomalies, we identify none of them. Instead, we decided to provide only localization service, where the actual diagnosing workload is left to the system designers.

We expected to compare traces in an A-B testing style. It turned out that attributes are very helpful in this case, where we can find out in detail similarities and differences between traces showing different symptoms. Moreover, the time-related attributes now became less important, because they instead function as a signal for us to trigger traces, rather than as an attribute that marks the difference between traces.

Based on the new approach, we formulated our output as a prioritized list, where every entry is a set of attributes that provide localization information. What remained was an algorithm to construct such a list.

8.3 Algorithm: From Bayesian to Association

Our initial choice of the algorithm was derived from the Bayes' Theorem, which calculates the probability based on the prior knowledge related to the current event. The idea seemed to fit our goal nicely: based on the previous knowledge of traces, we should be able to infer the root causes.

However, after realizing that the Bayesian Approach requires not only information about traces but also information about previous errors (which sadly is impractical), and that the mere information of trace attributes was unable to provide a Bayesian analysis, we decided to adopt a new approach.

Prof. Vigfusson presented the idea of Frequent Itemsets and the Association Rule, which gave us hope for a new possible direction. Generally speaking, the thought to find a connection between attribute sets and an anomaly is very similar to finding the association between a set and an item. But we need to give more careful definitions of what an attribute set is and more importantly, how to model itself as an "item". The result is the anomaly attribute. Unlike Hindsight, which treats anomaly merely as a trigger signal, we give it more emphasis to make it a part of the trace data.

Moreover, unlike the data used in the Association Rule, our attributes are more structured. Specifically, we can group traces by either the attribute key or value or even both. Therefore, additional efforts were spent to theorize our modified algorithm to replace the original one to make it actually work in our case.

8.4 Implementation: Hindsight and More

The decision for implementation upon Hindsight came from the hope to reduce the coding workload. However, it turned out that Hindsight itself was more complicated than what was described in its paper. The underlying architecture included more design details that were not specified and therefore much effort was spent on fully understanding its code.

Implementation was not easy either. Although the wrap function we wrote for generating attributes is quite straightforward, other components required more coding workload. The attribute database borrowed ideas from Prof. Mace's design of the doubly linked list; the query functions were written from scratch; the association algorithm was implemented through a modified a priori algorithm.

8.5 Takeaways

1. The production of our program arises from repeated group work of deciding the goals, formulating the theories, implementing the designs, and testing. At first, we only had vague ideas about what we expected to achieve. As we considered it more thoroughly in example scenarios of different complexities, we grew more clear of the future directions.

2. The Bayesian approach theoretically fits our goal but in practice was unlikely useful given the underlying data structures. Practical feasibility plays an important role in our choices of theories and algorithms.

3. The design of our program relies on the existing ideas. It was built upon Hindsight for its retrospective sampling. It also borrowed ideas from many other projects and theories and made modifications to fit our case.

Chapter 9

Conclusion

The increasing complexity of the distributed system poses a great challenge for diagnosis when anomalies occur. While there are existing projects making root cause analysis in the distributed system, they were all limited in some ways. Our project takes a different approach: we focus on localizing problems rather than identifying them.

To address the goal of automatically detecting and localizing anomalies that occurred in a distributed system, we build an architecture that uses the attributes as the underlying data structure to aggregate traces and the modified Association Rule for analysis It produces a prioritized list for every triggered trace to provide real-time anomaly localization, without providing the actual root causes.

Bibliography

[1] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating {Root-Cause} diagnosis of performance anomalies in production software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 307–320, 2012.

[2] The Jaeger Authors. Jaeger: Open source, end-to-end distributed tracing, 2017. URL `https://www.jaegertracing.io/`.

[3] The Zipkin Authors. Openzipkin: A distributed tracing system, 2017. URL `https://zipkin.io/`.

[4] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[6] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Differential provenance: Better network diagnostics with reference events. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2015.

[7] 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)*, 2007.

[8] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*, pages 34–50, 2017.

[9] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 243–254, 2009.

[10] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S Gunawi, and Shan Lu. Dfix: automatically fixing timing bugs in distributed systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 994–1009, 2019.

[11] Shan Lu, Haopeng Liu, Guangpu Li, Haryadi Gunawi, Chen Tian, and Feng Ye. Automatically detecting distributed concurrency errors in cloud systems, March 24 2020. US Patent 10,599,551.

[12] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 378–393, 2015.

[13] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[14] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd, 2019.

[15] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[16] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. *ACM SIGCOMM Computer Communication Review*, 44(4):383–394, 2014.

[17] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 395–420, 2019.

[18] Lei Zhang, Vaastav Anand, Zhiqiang Xie, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing edge-cases in distributed systems, 2022. URL `https://arxiv.org/abs/2202.05769`.

[19] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Distributed time-aware provenance. *Proceedings of the VLDB Endowment*, 6(2):49–60, 2012.