Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Distributed Tracing-based Adaptive Fault Diagnosis in Distributed Systems

Scientific Students' Association Report

Author:

Regina Bodó

Advisor:

András Földvári
Dr.Imre Kocsis

2023

Contents

Kivonat i

Abstract ii

1 Introduction 1

1.1 Structure of the report . . . 2

2 Distributed Tracing 3

2.1 The demand for distributed tracing . . . 3

2.2 Complexity of Microservices Architecture . . . 3

2.3 Tracing for Distributed Systems . . . 4

2.4 Evaluation of Traces . . . 5

2.5 Dashboard Solutions . . . 6

2.5.1 Problems and Solutions . . . 6

2.5.2 Potential downsides . . . 6

3 Diagnosis in Distributed Systems 7

3.1 Diagnostic Approaches . . . 7

3.2 Probing . . . 8

3.2.1 Pre-planned probing . . . 8

3.2.2 Active probing . . . 9

4 Pre-planned probing with distributed traces 10

4.1 Mapping Traces to Probes . . . 11

4.2 The construction of minimal diagnosis sets . . . 12

4.2.1 Selection Algorithms . . . 13

4.2.2 Selecting the minimal probe set for fault detection and diagnosis . . 14

4.3 Fault diagnosis with distributed traces . . . 15

4.4 Limitations of the approach . . . 16

5 Active probing with distributed traces 18

5.1 Single fault location with the use of error tags 18
5.2 Single fault location in the case of Fault hiding 20

6 Summary **24**

Bibliography **25**

Kivonat

A felhőszolgáltatások és az elosztott rendszerek növekvő népszerűségével a vállalatok új és más típusú kockázatokkal szembesülnek a rendszereik rendelkezésre állását illetően. Ezekben az egyre összetettebb környezetekben nehezebbé válik a rendszer bonyolultságát figyelembe venni, és a rendszer állapotára vonatkozó információkat hatékonyan kinyerni.

A dolgozat bemutat egy adaptív diagnosztika alapú hibadetektálási és hibadiagnosztikai megközelítést komplex elosztott mikroszolgáltatás alapú rendszerekben. A klasszikus adaptív diagnosztikán túlmutató új elem az elosztott rendszerekből származó működési nyomvonalak által szolgáltatott információ integrálása a diagnosztikai folyamatba.

Az adaptív diagnosztika célja, hogy a diagnosztikai szondákkal történő hibadetektálást és hibalokalizációt megkönnyítse, gyorsabbá és hatékonyabbá tegye a már létező korábban használt megoldásokhoz képest. A hatékonysága abból a módszerből ered, hogy a diagnosztikai szondákat fokozatosan választjuk ki, ahogyan információt nyerünk a rendszer állapotáról. Ez a hagyományos megközelítés azonban egyre nagyobb kihívást jelent a modern rendszerek összetettsége és a virtualizáció elterjedtsége miatt.

Az elosztott rendszerek nyomkövetése világosabb és pontosabb képet adhat a rendszer állapotáról és hibáiról. Ezenkívül megkönnyítheti a diagnosztikai folyamatot azáltal, hogy értékes betekintést nyújt a rendszer folyamataiba. A nyomvonalak az adat és vezérlés-áramláson túl tartalmazzák a komponensekre vonatkozó állapotinfromációkat, és ezek felhasználásával az adaptív diagnosztikai módszerek hatékonyabban alkalmazhatók.

A dolgozat célja az elosztott nyomkövetésből származó információk integrálása a diagnosztikai folyamatba, kiegészítve ezzel a klasszikus adaptív diagnosztikai megközelítést. A dolgozat, a mintapéldákon túl, egy mikroszolgáltatás referenciarendszerben mutatja be a megközelítést, tárgyalja annak előnyeit és kapcsolatát a meglévő megoldásokkal.

Abstract

With the rising popularity of cloud services and distributed systems, companies face new and different types of risks regarding the availability of their systems. In these increasingly complex environments, it becomes more difficult to account for the intricacies of the system and extract information about its state effectively.

This report presents an adaptive diagnostics-based approach to fault detection and diagnosis in complex distributed microservice-based systems. A new element beyond classical adaptive diagnostics is integrating the information provided by operational traces from the distributed systems into the diagnostic process.

The effectiveness stems from progressively selecting diagnostic probes as information about the state of the system is obtained. However, this traditional approach is becoming increasingly challenging due to the complexity of modern systems and the prevalence of virtualization.

Tracking distributed systems can give a more transparent and accurate picture of the system's state and faults. It can also facilitate the diagnostic process by providing valuable insight into the system's processes. Traces include status information on components in addition to data and control information and can be used to apply adaptive diagnostic methods more effectively.

This report aims to integrate information from distributed tracking into the diagnostic process, thus extending the classical adaptive diagnostic approach. In addition to sample examples, the paper presents the approach in a benchmark microservice application, discussing its advantages and its relation to existing solutions.

Chapter 1

Introduction

Diagnostic methods usually include a great amount of preparation in the form of creating models or constructing appropriate probes. Many resources could be spared if the diagnostic approaches could use pre-existing setups and tools already at hand and integrated with our system. Tracing is becoming a fundamental element of distributed system monitoring, considering the ever-growing complexity of the applications and the abstraction of the environments they are getting deployed to.

This report presents an adaptation of the active probing diagnosis method [6]. In the presented approach, we want to integrate the additional information and context provided by distributed tracing to be able to gain insight into the systems more effectively. The report presents the following contributions to the diagnostic process of the distributed traces (Fig. 1.1):

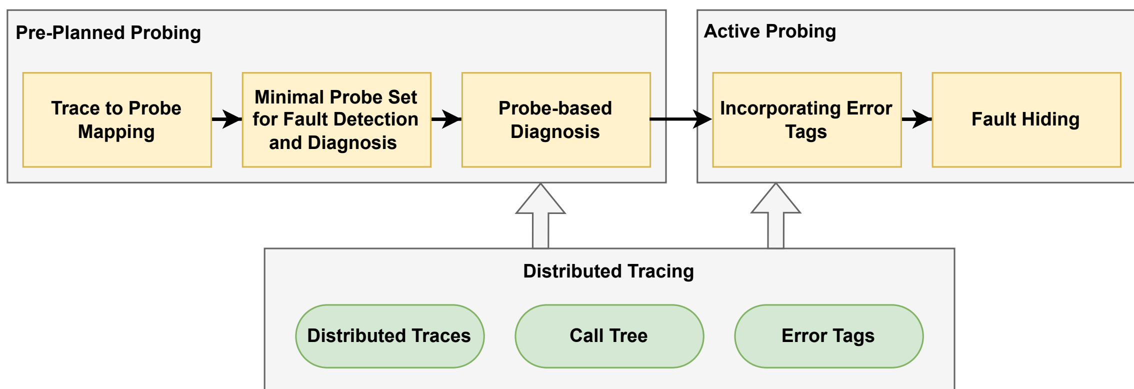


Figure 1.1: Pre-Planned and Active Probing for Distributed Traces

- **Trace to Probe Mapping:** *Traces from distributed tracking can be mapped into diagnostic probes.*

  If the traces meet the fundamental structural standards of the probes and are convertible to one another, then what has been stated previously will also apply to the transactions generated by the traces.

- **Minimal Probe Set for Fault Detection and Diagnosis:** *The minimum diagnostic probe set can be selected from the mapped diagnostic probes.*

After applying the structural constraint the transactions will remain functionally equivalent. Therefore, the goal is to construct a minimal fault detection and diagnostic probe set that can be utilized just like a regular probe set.

- **Probe-based Diagnosis**: *The minimum diagnostic probe set from distributed traces can be used for fault diagnosis.*

  The report shows that the transformed minimal probe set can be used for diagnosis as for the classical distributed probing approaches for diagnosis.

- **Incorporate Error Tags:** *The fault and call graph information extracted from the traces can be used to increase the accuracy of single fault diagnostics.*

  Metadata can be extracted form from traces like the call graph of the transaction. With this additional information, we are more capable of limiting the root cause of the fault. In this case, we expect spans of the trace to propagate the error they raise or receive.

- **Fault Hiding:** *The fault and call graph information extracted from the traces can be used to increase the accuracy of single fault diagnostics, even in the case of fault hiding.*

  The report introduces a new type of marking approach for the traces. This way, the algorithm is able to evaluate information about the different components the transactions visit and compare them to draw conclusions regarding their relations and the potential source of a partially hidden fault.

1.1 Structure of the report

Section 2 Introduces the importance and methods of distributed tracing. It describes its main goals, possibilities, and drawbacks as one of the most commonly used monitoring methods in distributed systems.

Section 3 presents the basis of system diagnosis in distributed systems. It lists some of the used approaches along with their disadvantages. The topic of adaptive probing is mentioned as previously invented and described in the paper, Adaptive diagnosis in distributed systems[6].

Section 4 describes the solutions for the use of distributed traces in pre-planned probing. It demonstrates the construction and use of the probes created from traces.

Section 5 introduces the topic of adaptive probing with the use of probes transformed from traces. It presents a new type of annotation of traces to present them in a format that is processible by algorithms.

Chapter 2

Distributed Tracing

Distributed tracing is a method used in software systems to monitor and understand the flow of requests as they travel through various components of a distributed application. It provides insights into how different microservices or components interact and can be used to diagnose performance issues, detect errors, and optimize system behavior. Distributed tracing helps developers and operators gain visibility into complex systems.

2.1 The demand for distributed tracing

In the world of microservices and highly distributed systems, pinpointing the source of performance bottlenecks or errors can be challenging. Traditional monitoring tools often fall short in these scenarios.

In modern software development, where applications are built using microservices and are often deployed on cloud infrastructures, the need for distributed tracing has become increasingly evident. While these new methods and environments can increase efficiency and functionality, the complexity of our systems is increasing as well. Companies need a way to make this vast amount of information for engineers comprehensible. Distributed tracing is one of the best tools for this task.

2.2 Complexity of Microservices Architecture

Microservices architecture is a design approach in which an application is constructed from small, independent services that communicate over a network. Each service is responsible for a specific business capability and can be developed and deployed independently. This architectural style offers agility and scalability but introduces challenges related to monitoring and observability.

The benefits of microservices, such as agility and scalability, are clear. However, they come with a trade-off: the introduction of complexity. Requests often traverse multiple microservices, potentially running on different machines or containers, creating a complex web of service interactions. This complexity can make it challenging to track the flow of a request and understand the end-to-end journey.

To address these challenges, distributed tracing plays a pivotal role. It captures data at a granular level, breaking down each request into individual spans, each representing a specific operation or step in the request's journey. This granular visibility provides

Trace Component	Description
Trace ID	A unique identifier for the trace.
Span ID	A unique identifier for each span.
Parent ID	Links child spans to their parent span.
Timestamps	Recording when the span started and ended.
Tags	Key-value pairs providing additional context. e.g., errors
Logs	Records of specific events within a span. (extracted from component logs)

Table 2.1: Main Trace Components

a detailed view of how requests interact with different services, which is essential for comprehending and managing the complexity inherent in a microservice architecture.

2.3 Tracing for Distributed Systems

Tracing in distributed systems is a method for monitoring and analyzing the flow of requests and transactions as they move through different components and services in a distributed or microservices-based architecture. It helps developers and operators gain insight into the behavior and performance of these systems, making it easier to diagnose issues, optimize performance, and ensure reliability.

Table 2.1 presents the main components typically included by a trace. In distributed tracing, a trace represents the journey of a single request as it moves through different services. It includes spans, which represent a specific operation or a portion of the request's path. Spans are interconnected to form a trace, showing the entire flow of the request.

Spans are the fundamental building blocks of a trace. A span represents an individual operation within a service. It includes information like the operation name (e.g., "HTTP request," "Database query"), the start and end timestamps, and any relevant metadata.

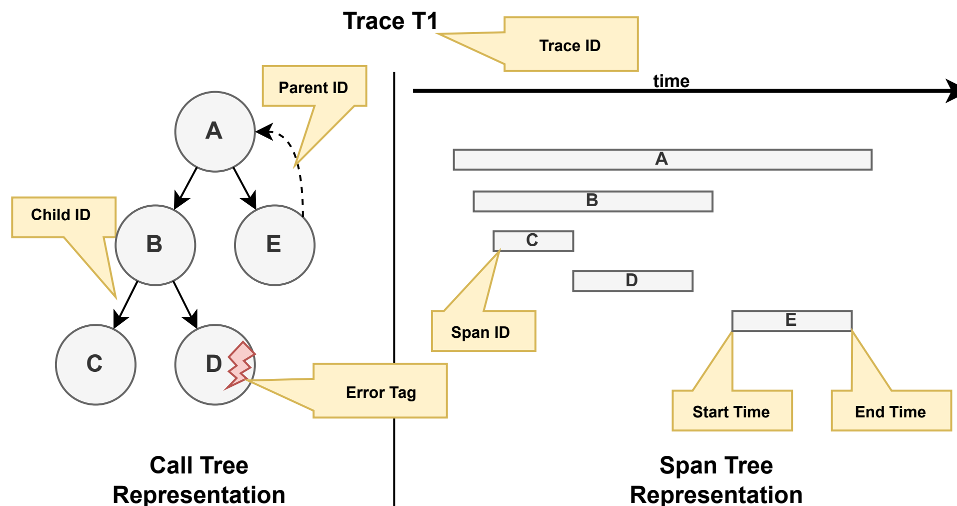


Figure 2.1: Call tree and span tree representation of a trace

Data tracing within your applications involves integrating code using instrumentation. This method allows for trace data to be collected. Typically, this integration is achieved through the utilization of libraries or software development kits (SDKs) that proffer application programming interfaces (APIs) for the establishment and supervision of Spans.

The open-source OpenTelemetry project [9] is a framework (collection of APIs, SDKs, and tools) designed to enable and enhance observability in modern software applications. It plays a crucial role in distributed tracing and performance monitoring. It offers support for various programming languages.

The process of distributed tracing involves generating trace IDs, span IDs, and trace flags, as well as recording timing data and metadata for each span. The goal is to instrument the code so that you can track the start and end of operations within your services.

Instrumentation can generate tremendous overhead if not used well. The usage has to be planned and estimated at the start of the project. Setting up tracing from scratch after a project has already been written can absorb a lot of both human and system resources.

Alternatively, we can use the auto-instrumentation tools that trace our applications through the pre-instrumented libraries and managed environments. These can be practical if we want to take a quick look at our gains and possibilities. However, they are restricted in many ways. There is a wide range of configuration settings we can use, although if we want something specific, it likely would not be achievable. This includes points for collecting information and also the type of metadata collected.

2.4 Evaluation of Traces

To make the most of distributed tracing, it's crucial to extract insights from traces [1]. This involves analyzing trace data to identify patterns, anomalies, and areas for improvement. Some common insights include request latency, error rates, and service dependencies.

One of the primary use cases of distributed tracing is root cause analysis. When a problem occurs, tracing data can be used to follow the path of a request and determine which service or component caused the issue. This significantly reduces the time it takes to resolve problems and improve system reliability.

To make the analyses effective, we need to be able to search between our traces. Luckily, most of the tools available provide us with the functionality. When we select a trace, it can be visualized (Fig. 2.1) as a call tree or a span tree representation. Different tools visualization can be slightly different, but the structure is almost exactly the same.

On the first look at Figure 2.1, we can see and identify the key information, like the trace ID and the duration of the whole trace. Next, we can inspect each span closer if we open the detailed view connected to them. From the tree structure of the trace, we can see how different spans relate to each other and which components were called. Another important sign to look out for is if any of the spans have an exception or an error in them. This could mean that the trace requires further investigation because there may be an underlying problem. We can also inspect the individual duration of the spans and compare them if we know what the average should be to identify hidden issues. We can also see the bottlenecks and hot spots of the call and see which part takes the most time, thus slowing down our response.

Other than the basic annotations in traces, we can use the previously defined metadata that we added to the traces during instrumentation. This data can be crucial as it is

tailored to the system's particular features. However, this also means its interpretation can also be unique, so it is probably worthwhile to maintain and consult documentation related to these specific parameters.

2.5 Dashboard Solutions

Dashboards are critical for visualizing trace data and making it actionable. They provide a central location to view and explore traces, which is vital for monitoring and troubleshooting. Dashboards often include features like filtering, aggregation, and alerting.

2.5.1 Problems and Solutions

Dashboard solutions (e.g., Prometheus [4], Jaeger UI[3]) try to actively manage the complexity of distributed systems. The amount of information can be vast; tools try to simplify them by providing clear visualization.

Dashboards provide real-time insights into the performance and behavior of various services and components within a distributed system. Engineers can see how different parts of their applications interact with each other.

They collect and aggregate performance metrics, such as response times, error rates, and resource utilization, allowing engineers to identify issues and trends quickly. These metrics can be stored and used to conduct studies covering a longer period, enabling engineers to make informed decisions about system optimization and resource allocation.

Often, they come with alerting capabilities that can notify engineers when certain performance thresholds or error rates are exceeded. This enables proactive issue resolution.

2.5.2 Potential downsides

Integrating and effectively using distributed tracing dashboards presents a multifaceted challenge. It is important to recognize that it can take engineers a significant amount of time to become proficient in their use, given their complexity.

The process of capturing and preserving trace data comes with a noteworthy consideration – it can impose an additional load on system resources. This may not be ideal, particularly for systems operating within strict resource limitations.

Moreover, it is crucial to be mindful of the financial aspect. Deploying specific distributed tracing solutions, particularly at a large scale, can entail substantial expenses. These costs encompass the resources required for infrastructure and licensing fees.

Another aspect to consider is the potential impact on privacy and security. When dealing with systems that handle sensitive data, the storage of trace data may give rise to concerns in these domains.

Lastly, it is important to exercise caution when configuring alerting mechanisms. Overly sensitive settings can result in a high frequency of false alarms, potentially leading to alert fatigue among the teams responsible for system management.

Chapter 3

Diagnosis in Distributed Systems

The increasing complexity of modern computing systems establishes the need for accurate and efficient diagnostic approaches. Some of the difficulties can be alleviated with the already-known classical [8] methods. However, these approaches all have their constraints, which make them less sufficient for handling the ever-changing nature of today's distributed environments. This chapter presents the methods described in the [6] for pre-planned and active diagnosis of distributed systems.

3.1 Diagnostic Approaches

One of the commonly used approaches for system diagnosis is event correlation. In this case, components are instrumented to send out alarms when their state changes. These changes are received by a centralized manager that correlates these events in order to identify the fault in the system.

The approach's accuracy comes with a great price as it requires heavy instrumentation in each of the components to equip them with the capabilities to send various signals indicating their current state. The conditions and type of the signals have to be designed manually. Despite all the effort put into preparations in the most crucial moments, when a component fails entirely, it may not even be able to send out an alarm before it shuts down. If our system contains 3rd party components or external system calls, we can not instrument them even if we wanted to.

One of the other widely available methods for diagnostics is end-to-end probing. This method consists of previously constructed transactions that are sent into the system. Each transaction travels through different components. The result of the transaction, whether it fails or succeeds, depends on the health of the contained components. The transactions are sent from dedicated computers called probing stations. These machines are responsible for scheduling and choosing the next transaction to run based on the previous settings.

Probing has its own limitations, too. Usually, the probes are constructed with only intuition based on experience and general rules based on practical examples. This could result in highly sub-optimal sets where most of the probes generate overhead but do not increase the gained information. Many of the transactions in large sets may cover problems that never eventually occur, so running them frequently is superfluous.

Both techniques possess the limitations of batch processing of the symptoms, which does not correlate with the preferred method of continuous and dynamic monitoring of the dynamically changing systems. The problem is that these methods work with static models

where they mainly detect "hard" failures. These are the types of faults that explicitly show that there is something wrong. Whereas the "soft" failures are harder to detect, such as longer response times.

3.2 Probing

Probes are pre-defined transactions sent to the system for processing. The components the probe contains are known. The probes' result depends on the state of the contained components. If one of them is faulty, then the whole probe will fail.

Probes are managed by a computer external to the system. Its main responsibility is to start predefined probing transactions and collect their results. They manage the timing and selection based on predefined rules.

The Dependency Matrix (Table 3.1) is a table containing the intersection of components (component fault modes) and the different probes. Each probe is represented as a row in the matrix. The columns represent each of the components the system consists of. A cell is set to 1 if the probe's run includes the component along its path, and 0 if it is not.

components	A	B	C	D	E
Probe a	1	1	0	0	1
Probe b	0	0	1	1	0
Probe c	0	0	0	1	1

Table 3.1: Dependency Matrix

3.2.1 Pre-planned probing

Pre-planned probing aims to greatly reduce the randomness in selecting the set of probes to be run. It constructs two different sets with different purposes and makes them as effective as possible. It achieves this by reducing the original set of pre-defined probes while it is still capable of achieving its goal.

The first one of the two sets is the set for fault detection. The goal of the fault detection set is to be able to indicate if any of the nodes failed. To achieve this, the set of transactions has to "cover" all nodes. Each of the nodes has to be contained by at least one probe. In paper [6], the authors call this decision problem Probe "*Probe Set Selection for Fault Detection*". The minimal detection set selection is an NP-hard problem as it is identical to the "*Minimum Set Cover*" problem, which is known to be NP-hard.

Although the problem is NP-hard, there are heuristic search algorithms that are simple and efficient. One of the examples is their active-probing algorithm, which will be described later. In a scenario where there are no faults in the systems, this algorithm won't stop until it has not validated the state of all the components.

The second set is the "*Minimal Set for Fault Diagnosis*". This set has to be capable of identifying the source of the fault from the result of its probes. The goal is to identify the smallest subset of probes that can identify the same set of faults as the original.

The paper states that the conditions for this case can be explained using the dependency matrix's notation. The smallest set of probes that can identify the same set of faults would have a dependency matrix where every row is unique. They formalized this decision

problem and called it "*Probe Set Selection for fault Diagnosis*". They prove that the problem is NP-hard via reduction from "*3-Dimensional Matching*".

Even though this problem is NP-hard, there are algorithms with polynomial-time approximation that perform well in practice. Two of these algorithms are presented.

The greedy search algorithm (Alg. 1) aims to build the minimal set by adding transactions one by one. It starts with an empty set and adds the probe that increases the information about the system the most. It stops when there aren't any more probes that could be added to the set.

Algorithm 1 Probe-Set Selection: Greedy Search

Input: A set of available probes **T**
Output: A subset of probes
Initialization: The output set is empty.
do
1. select most informative probe
2. update the probe set

while The selected probe updates the information about the system
Return The final subset of chosen probes

The calculation of the amount of gained information is based on probability. It is assumed that only one component fails at a time, thus reducing the complexity of the required calculations.

Subtractive search starts with all the known probes and removes them one by one. A probe is kept if its removal results in the loss of information regarding the system's state.

3.2.2 Active probing

The goal of active probing is effective fault localization. It proposes the idea of selecting the probes adaptively. This way, we don't use more probes than we need, thus reducing the number of required probes to run significantly. This is achieved by the fact that this algorithm stores and updates the belief about the system's state after every run transaction. This raises the efficiency because the next probe is chosen according to the most relevant and up-to-date context we have. The next probe is chosen by the idea of maximizing the information gained if the probe is run. It has the advantage of having the latest information about the system's state. The greedy algorithm lacks this because it can only make assumptions by averaging the possible states.

The inputs are the set of all the available probes. It maintains the current beliefs about the system's state and updates it every time a probe was run. It selects and runs probes until no more information can be gained from running any of them. That's when it stops, and the output is the set of probes run.

Chapter 4

Pre-planned probing with distributed traces

The adaptation of Active probing [6] utilizes the information contained in traces. We will see that the structural and logical conversion between traces and probes is viable and sufficient.

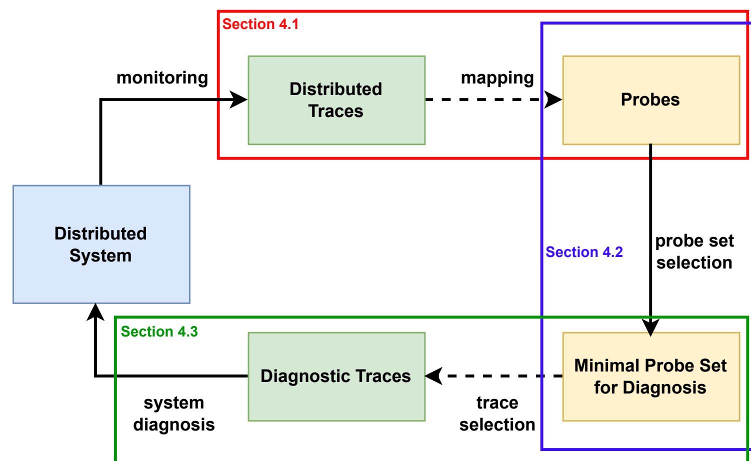


Figure 4.1: Pre-planned probing with distributed traces

Figure 4.1 represents the different parts of the process of adapting the traces. The Distributed traces are received from monitoring the system. These traces are then mapped to resemble the structure of probes. The goal is to get traces into a form where they are usable in the usual probing algorithms. The methods and processes are further specified in Section 4.1.

With the transformed traces, we construct the minimal sets used for fault detection and diagnosis. These sets prove that the transform traces not only adhere to the structural constraint of probes but with the previously demonstrated algorithms, the minimal sets are constructible from them. The statement is further discussed and represented by examples in Section 4.2.

The minimal sets are used in an example to represent the functionality of the created minimal sets. The examples are described in detail in Section 4.3.

In real scenarios where the application is deployed in a cloud environment. it can be difficult to investigate physical failures or problematic deployment scenarios. The reason

is that the location of the exact replicas of each of the components is hard to predict. The shared nature of the resources also poses a great threat to reliability. For example, another user can overload the machine that our application is deployed on, so our resources become scarce, too.

In our processes, we will treat distributed traces as general transaction types. This can be achieved in reality by filtering traces by time until the results from matching transactions in a given time range match between traces.

4.1 Mapping Traces to Probes

In this section, it is achieved that the traces conform to the normal structure of probes. This way, the operations previously applied to probes can be applied to the transformed traces as well. On the basis of the information discussed below, the following is stated:

Traces from distributed tracing can be mapped into diagnostic probes.

The first objective is to see if traces can follow the structural representation and behavior of probes. Probes and traces partially differ in structure and the way they represent and contain information. This could affect the applied techniques slightly. Therefore, it is important to discuss the changes compared to the original circumstances. This list presents the main differences between probe and trace-based diagnostics.

- **Traces are based on preexisting data, unlike probes that have to be run.** Traces are collected continuously while the system is operating. This way, in contrast to probe-based diagnosis, we don't have to run probes on purpose; rather, already existing data is available.
- **Transaction-based probes do not have to be defined.** The step to define probing methods is eliminated by the fact that already existing transactions cover most of the user interactions, use cases, and components.
- **Many transactions relate to the same trace.** Transactions can have many related traces of the same type. These traces can be filtered by time or transaction type. The filtered traces can be used for the adaptive diagnosis.
- **Traces contain extra information.** Traces come from the monitoring of the application. This data is used to gain insight into the components' health, including the error type, call sequence, and metadata.

In this stage, the traces are transformed in the most simple and straightforward way. In the solution, the following constraints are applied: The transactions defined by the users (Fig. 4.1) are always traveling on the same path, thus crossing and containing the same components. Transient faults are not taken into account, which means in traces belonging to the same transaction type, the appearance of faults is uniform. According to this hereinafter, instead of referring to single traces, they are represented by their transaction type.

Here, we say that we will consider every transaction we have as a possible probe. Every component that gets called in the traces is considered like a component that's crossed and

is thus tested and checked by a probe. This way, we can build a dependency matrix from the transactions that resemble the dependency matrix from [6].

In this case, one row of the matrix represents a sample trace, and the columns represent each distinct component of the distributed application. The 1-s in the matrix represents that the trace utilizes that component, and it appears in the trace's call tree.

On Table 4.1, we can see a demonstrative example for this type of dependency table.

components	A	B	C	D	E
transaction a	1	1	0	0	1
transaction b	0	0	1	1	0
transaction c	0	0	0	1	1

Table 4.1: Dependency Matrix from traces

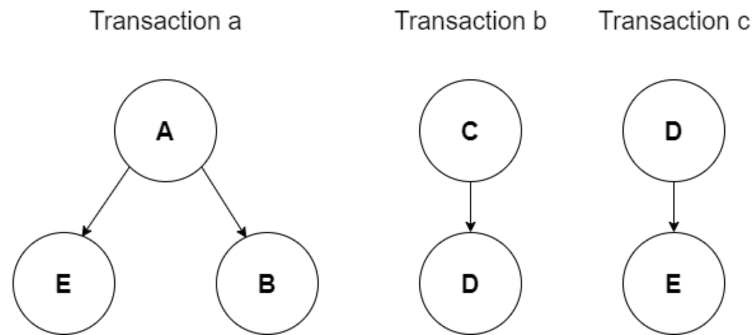


Figure 4.2: Introduction

4.2 The construction of minimal diagnosis sets

The following section demonstrates the application of minimal set construction on transformed transactions. According to the result of the represented examples, the following is stated:

The minimum diagnostic probe set can be selected from the mapped diagnostic probes.

The assumptions applied for this section can be seen in Table 4.2.

Section	Single / Multi fault	Fault hiding	HW faults
Minimal set construction	Single	False	False

Table 4.2: Diagnostic Assumptions

Although we can see that our transformation results in assets similar to the ones used in the original adaptive probing, the next question is if they live up to the same functionality as the originals. To achieve this, we test if a minimal detection and diagnostical probe set is constructible. After this, we will have our sets and see if they are able to detect and localize the fault the same way in case of a failure.

4.2.1 Selection Algorithms

Similarly to the paper[6], I want to select the minimal set of probes to detect and diagnose faults in the system. The starting point for this is the previously transformed transactions. We want to validate their usability by being able to create minimal sets and use them.

The fault detection set should be able to fail if any component fails. We want to find the smallest set of rows in the dependency matrix that each of the columns has at least one non-zero entry.

This problem matches the Minimum Set cover problem, which means it is Np-hard. However, there are suitable heuristic approaches that are simple and efficient enough for us.

The task of **fault diagnosis** is distinguished from detection. In the case of diagnosis, we have to be able to identify the result the probes returned and locate the problem. The goal is to reduce the number of probes but still be able to detect the same faults as the original.

In the paper[6], they discuss that this problem can be reduced from 3-Dimensional Matching, which means it is Np hard. Despite this, efficient polynomial-time approximation algorithms exist, which can be used to determine the minimal set for fault diagnosis. Two such algorithms are greedy and subtractive search.

Greedy search starts with an empty set and adds the probes progressively. It decides which probe to add depending on the increase in information it gives. The next probe added is the one that provides the most of previously unknown information. Once there are no more probes that can increase the information about the system, the algorithm stops.

Subtractive search starts with all the probes and iterates over all of them. If Removing the probe does not result in information loss about the system, the probe can be omitted from the final set. Neither of the algorithms is optimal. Without any inner knowledge of the system, the use of them is ranked equally in effectivity. However, considering we have minimal knowledge relating to the transactions we are working with, slight optimizations can be made.

If we have lots of different kinds of transactions that use the same components, this could mean that the greedy algorithm is optimal because we can skip through similar transactions really fast, whereas in this situation, with the subtractive algorithm, we would have to remove all of these separately.

Another situation can be that the use cases are really different, and each of them is represented in one transaction. In this case, we expect that there aren't many probes with redundant information so that we can find the few quickly and effectively with subtractive search. Whereas with the greedy algorithm, we would have to add and evaluate most of the transactions separately.

components	A	B	C	D	E
transaction a	1	1	0	0	1
transaction b	1	0	1	1	0
transaction c	0	0	0	1	1
transaction d	1	0	1	0	0
transaction e	1	1	1	0	0

Table 4.3: Dependency Matrix

4.2.2 Selecting the minimal probe set for fault detection and diagnosis

A slight difference can be seen between the transformed table and the usual dependency tables. The transactions can quite scattered between components. It usually can not be said that there is one exact trace to detect all faults in the system. We can only construct a possible minimal set with one of the algorithms introduced previously.

For the example construction we will use the previously created dependency matrix based on the transformed transactions seen in Table 4.4.

components	A	B	C	D	E
transaction a	1	1	0	0	1
transaction b	1	0	1	1	0
transaction c	0	0	0	1	1
transaction d	1	0	1	0	0
transaction e	1	1	1	0	0

Table 4.4: Dependency Matrix

An example could be in this case the use of the Greedy algorithm. We can see that most number of modules a transaction can cover is 3. We choose the first one in this category, transaction a. Then we check which components are left uncovered, C and D. We check if, from the other rows, we can find one that covers both. Fortunately, there is one, transaction b. This completes the minimal set of transactions for fault detection as it can be seen in Table 4.5.

components	A	B	C	D	E
transaction a	1	1	0	0	1
transaction b	1	0	1	1	0

Table 4.5: Minimal Detection Set

A fault-detecting minimal set also needs to be created. To create this I used the same algorithm that was mentioned previously. We take each transaction and add it to our set if it brings extra information. If adding the transactions doesn't add any information we leave it out.

We consider a demostrative example to show how this would look like in action. The maximum number of components our transactions can cover is 3. There are multiple ones that cover this many so we inspect them in the same order as they come in. The first of the 3 component transactions is a. We add it to our previously empty starting set. Because the set was empty beforehand it seems evident that adding a will increase the amont of possible information we are able to gain on from the system. The next one is transaction b, it can be seen in Table 4.4, that b covers the modules C and D which were not covered by a. This means b adds useful information ao we add it to the final set. Transaction c is where the process gets more interesting.

The set contains transactions covering all of the components. For fault diagnosis the next requirement is to be able to choose any 2 of the components so that they are covered by different probes so this way it can be determined which one of them is the source of the fault and which is not based on the probes.

This is a heuristic method that simplifies the original and is applied for demonstrative purposes. The result is suitable to illustrate the main process, however it does not aling with the real algorithm in every way.

It can be seen in Table 4.4 that with only transaction a and b we can not satisfy this constraint. Both B - E and C - D component form pairs that can not be covered by different transactions right now so we need to add more to our set.

Transaction e can be added next, because it is the next one that covers 3 components. We can see that e covers component B but not component E. This means that the state of B and E can be examined separately. Before this was not possible. This means transaction e adds information so it is added to the final set. The component pair C - D still can not be handled, the algorithm continues.

The only transactions left are the ones that only contain 2 components. They will be used next. Transaction c is added next. It covers D from the component pair C - D. This way C's and D's status can be differentiated. This was not possible before, this means c brings possible additional information, therefore it is added to the final set.

With this the previously defined problematic component pairs were eliminated. Hence the algorithm stops. The resulting minimal diagnostic transaction set is represented in Table 4.6.

components	A	B	C	D	E
transaction a	1	1	0	0	1
transaction b	1	0	1	1	0
transaction c	0	0	0	1	1
transaction e	1	1	1	0	0

Table 4.6: Minimal Diagnostic Set

4.3 Fault diagnosis with distributed traces

The minimum diagnostic probe set from distributed traces can be used for fault diagnosis.

This section presents an example of fault diagnostics with the use of the previously defined sets to verify their use. I will use the previously defined minimal sets from Table 4.5 and Table 4.6.

As stated previously in this chapter, we consider the traces to propagate the error they receive this means that in the end if the whole traces fails correspond to when a probe fails during diagnostics. For this we will add an extra column to our dependency table which represents if the traces failed or not. In an average situation this would mean that we know at the start of the process which transactions failed an which did not. However in this case we want to see in these transactions could behave as regular probes so the information is hidden until that particular probe is choosen to be ran by the algorithm.

The extended matrix is presented in Table 4.7,

Right now everything is marked with a dash (**-**). This means that there is no information. In the Fault column a **1** will represent that the probe failed and return an error and a **0** would represent that the probe completed successfully and the related components are healthy.

components	A	B	C	D	E	Fault
transaction a	1	1	0	0	1	-
transaction b	1	0	1	1	0	-
transaction c	0	0	0	1	1	-
transaction e	1	1	1	0	0	-

Table 4.7: Extended Dependency Matrix

The following situation plays out. We get a notification that the detection set failed and there's a fault that need to be diagnosed. This means that transactions a and b have a result now. The result is represented in Table 4.8

components	A	B	C	D	E	Fault
transaction a	1	1	0	0	1	1
transaction b	1	0	1	1	0	0
transaction c	0	0	0	1	1	1
transaction e	1	1	1	0	0	-

Table 4.8: Fault detected

Inspecting the result of the failed detection set it can be seen that transaction a failed and transaction b succeeded. The next probe is chosen based on this information. Transaction a's failure means the fault is in components A, B, or E. Component A can be ruled out based on the fact that transaction b contains A too, and transaction b did not fail, therefore component A has to be healthy. Components B and E remain. There is insufficient information to determine if B or E is the component with the fault. Consequently, a new probe has to be run from the minimal diagnostic set. Transaction c is the next one and it contains one of the components in question so it suits our needs.

Transaction c is run and it failed. The result can be seen in Table 4.8 . The components it contains are D and E. Component D can not be faulty, because transaction b contains it and it succeded. This means that only E has to be faulty. Because we assume that we can only have single faults in our systems, every component except E has to be healthy. The fault is found so the algorithm stops.

It can be seen that the constructed minimal sets are capable of detecting faults like regular diagnostics sets.

4.4 Limitations of the approach

In the case on unfortunate events like programmers following best practices and succeeding in separating modules effectively. In a more real life scenario we could have a transaction table like the on on Table 4.9.

components	A	B	C	D	E	F	G
transaction a	1	1	1	0	0	0	0
transaction b	1	0	0	1	1	0	0
transaction c	1	0	0	0	0	1	1
transaction d	1	0	1	1	0	0	0

Table 4.9: Dependency matrix

There could be many cases, where traces don't overlap enough to use the original active probing. For example if trans. c would be faulty, the state of components F and G can not be differentiated.

One of the solutions could be the construction of artificial transactions with a narrowed-down set of included components. LinkedIn uses this method as their implementation of chaos engineering as described in the book, Chaos Engineering System Resiliency in Practice[7]. The method is that the system's components have built-in default and mock responses. These are enabled by specific parameters inside the requests and request cookies.

In the previously described situation, I would utilize them the following way. To differentiate the state of components F and G I would set the request that either F or G has to return a mock response. This way the component with the mock won't be tested and the other component's status can be inspected in isolation. This method is not discussed further, additional information can be gained from the book [7].

The other solution is the further decomposition of traces based on the metadata contained in them. The goal is to use up the extra information traces contain compared to probes so that our methods can be used in more general situations and still provide effective results. In the next section, I present one of the possible methods for achieving this.

Chapter 5

Active probing with distributed traces

Integrating trace information into the diagnostic process.

This chapter discusses what information can be extracted from traces to make them a more effective fault diagnosis. The first and most straightforward way traces have more to offer is by visually evaluating them. They are composed of many different spans. From these spans, the call tree of the spans can be reconstructed and used to gain further information.

One of the most relevant attributes of the spans is that they may contain an error-tag. This is useful because a well-constructed trace will be annotated with error tags in its span when something went wrong with the operation it is intended to represent. In the best-case scenario, the trace only contains an error-tag where the real fault is and the source of the problem is located at first glance. However, in real systems, that is not the usual case. Many components propagate or hide errors they receive or raise.

5.1 Single fault location with the use of error tags

The fault and call graph information extracted from the traces can be used to increase the accuracy of single fault diagnostics.

The assumptions applied for this sections can be seen in Table 4.2.

Section	Single / Multi fault	Fault hiding	HW faults
Propagation	Single	False	False

Table 5.1: Diagnostic Assumptions

That is the reason why we separate and handle these cases separately. This section discusess the case where every component that raises or receives an error, will surely contain an error-tag in its related span and propagate the error to its parent span and eventually to the root of the trace. The discussed method only includes situations where only one fault is present in the system. Multifault scenarios are not handled in this part.

To visualize this, we introduce a new table along with the dependency matrix. we will call this the error matrix. This table is similar to the Dependency matrix; each row represents a transaction, and each column is linked to one of the system's components. The difference

between them is that the error matrix represents different information and uses a slightly different marking system. The error matrix is used to display the presence or absence of the error tag in a span. Each of the spans in the transactions represents a call to a component. If the span contains an error-tag in a transaction, the span's component will have the error tag marked in its column with the number **1**.

If the transaction contains the component's span, but the span does not contain an error tag, the transaction will have a **0** in the components column. If a transaction does not contain a span related to a component, the transaction will have a **-** mark in the component's corresponding column.

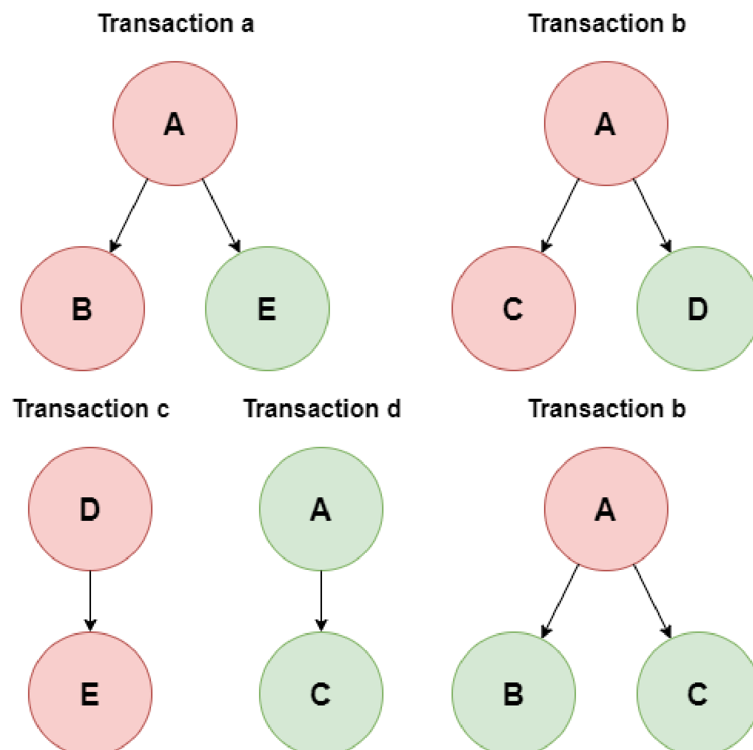
An example representing an example like this can be seen in Table 5.3.

components	A	B	C	D	E
transaction a	1	1	0	0	1
transaction b	1	0	1	1	0
transaction c	0	0	0	1	1
transaction d	1	0	1	0	0
transaction e	1	1	1	0	0

Table 5.2: Dependency Matrix

components	A	B	C	D	E
transaction a	1	1	-	-	0
transaction b	1	-	1	0	-
transaction c	-	-	-	1	1
transaction d	0	-	0	-	-
transaction e	1	0	0	-	-

Table 5.3: Error matrix



We can inspect Table 5.3 to further understand the possible notations. Transaction a goes through components A, B, and E. It does not include components C and D, so it has the mark **-** in the corresponding columns. The span of the call to component E did not contain an error tag, so it has the number **0** in the corresponding column. Component A and B's spans both had an error tag, so they are marked with the number **1**.

In the case of fault propagation, the location of the fault can be determined by a simple algorithm. The algorithm was introduced in [5], where the authors use the approach to localize faults in the cloud environment.

The algorithm utilizes the call tree of the transaction. It iterates over the nodes of the tree according to the following rules. We only iterate over components that have an error-tag. Due to the absence of fault hiding the faulty component's span must contain an error tag. Otherwise, it would be useless to inspect the components' status further. We start with the root span of the trace. Since this is the origin of all the other calls, it will necessarily have an error-tag in case of a fault. The next step is to examine the faulty node's children. If the node is faulty and does not have any children nodes, we find the cause of the error. Considering the assumptions in this section, the error must have been raised by the component itself. If the node has children and they are not faulty, we have also found the cause for the same reason as before. If any of the children have an error tag, we continue the iterations by stepping over to the faulty child and marking it as the main inspected node next.

An example is demonstrated of this algorithm on one of the previous call trees. In transaction a we start with component A, that's the root component and it has a fault. A has 2 children, B and E. E has no error-tag so it can be skipped. B has an error tag so B will be our next inspected component. B does not have any children so

This algorithm can already show great results if used properly as described in [5]. However, it does not include and handle many of the real-life scenarios engineers face every day. In the next section, we explore the possibilities in cases where components may cover up their faults and do not necessarily propagate them to their parent components.

5.2 Single fault location in the case of Fault hiding

The section describes the way to annotate traces so that the information they hold can be extracted in a structured manner. As a consequent result of this, algorithmic methods are capable of producing diagnostics results based on the provided information. Based on the systems and methods presented below, we claim the following:

The fault and call graph information extracted from the traces can be used to increase the accuracy of single fault diagnostics even in the case of fault hiding.

The starting trace failed, so it must contain an error-tag in one of its spans. The first part of the method aims to inspect if the fault is closer. It selects traces that include the components that failed in the previously inspected trace.

In most cases, some components will hide the fact that they received an error and try to handle the situation by resulting in some type of fallback. The status of the traces are represented with a special annotation to represent this ambiguity. These annotations can be expressed in an error matrix (e.g., in Table 5.5), where the T_{ci} represents the annotation for the i. trace of the component c.

Our annotation system contains four different marks. Shown in Table 5.4

mark	meaning
0	Healthy component. (span has no error tag)
1	Faulty component. (span has error tag)
?	Unsure, the component is Suspicious.
-	Unknown.

Table 5.4: Marks and their meaning

- **0** means that the span associated with that component doesn't contain an error-tag. We suspect that the component is healthy.
- **1** means that the span associated with that component contains an error-tag. We suspect that the component is faulty in some way.
- **?** means that the component precedes another component with an error in the call tree. That means that this component's data is used later in a different component, so this component may be faulty in a way that propagates to another one.
- **-** means that the trace didn't contain that particular component, so we don't have any additional information.

We use these to compare traces. The result of the comparison is always the stronger notation. The strength order of the notations is as follows:

$$\textbf{-} < \textbf{?} < \mathbf{1} < \mathbf{0}$$

The final diagnostic result of components is the maximum of all the annotations from all the traces relating to that particular component. The diagnostic vector D is constructed from the diagnostic result of each component in the following way:

$$D =< D_{c1,\ldots D_{cn}} > \text{, where } D_{ci} \text{ is the diagnostic result for component } c$$

The diagnostic result can be calculated by selecting the maximum for all the annotations from all the included traces:

$$D_{ci} = max(T_{ci})$$

Figure 5.1 depicts the selection and evaluation of traces in the case of fault hiding. First, we detect an error from our traces. Then, they are presented in the error matrix with the annotations described later. The failed trace is the fault detection trace set.

In the next step, the next trace is selected to be evaluated. The selection is based on a heuristic method and aims to gain the most information possible. The first traces selected are the ones in the fault detection set. These traces are evaluated, and the result is added to the final diagnostic vector.

Each component's result is determined by the maximum of all the annotations of that component in all the inspected traces so far. When a new trace is evaluated, its result is added to the set of annotations. The result is reevaluated, and the diagnostic vector is updated.

Once the update happens, it is checked if it made changes to the previous diagnostic vector. If it does not change, then the algorithm is finished, and we cannot gain more

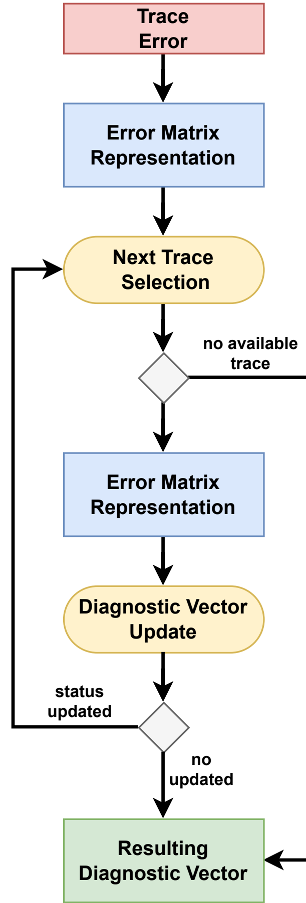


Figure 5.1: Adaptive diagnosis with distributed traces - Workflow

information. If it changes, the process starts over with the selection of a new trace. The next trace that is chosen is the one that covers most of the components that have the annotation **1** in the diagnostic vector.

After we don't have any more components containing the notation **1**, the aim is to eliminate the question mark notation from all the possible components. The trace is chosen, which contains most of the components that are marked at the moment with the question mark annotation.

The process continues until there are no more suitable traces to choose from or if no more information can be gained.

The algorithm is shown in a demonstrative example with traces from Table 5.5. The notations were determined by the call trees of the transactions shown in Figure 5.2 .

components	A	B	C	D	E	F	G
transaction a	0	1	?	-	-	?	-
transaction b	0	-	-	0	0	-	-
transaction c	0	-	-	-	-	1	?
transaction d	0	0	0	0	-	-	0

Table 5.5: Error Matrix

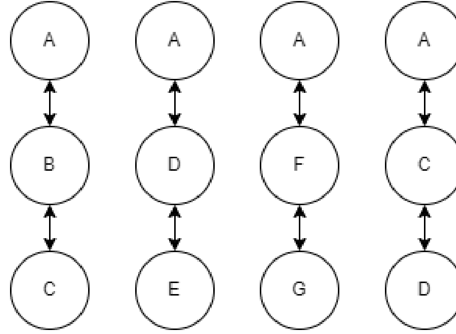


Figure 5.2: Call tree of example system

In this case, we find that transaction a has failed. We got an error-tag in the span that represented the call to component B. Component C and F didn't contain an error-tag but their calls precede the call to component B so they are a suspect for the source of failure.

Transaction a is evaluated and added to the final vector. Because the final vector is initialized previously with the weakest marks, the diagnostic vector now looks exactly like transaction a. The diagnostic vector changed during this iteration so we continue with the next trace. The algorithm chooses the next transaction to cover the number **1**'s in transaction a. Transaction d satisfies this so it is the next trace to be processed. Transaction b's perception of the components state is added to the information set, and then the final diagnostic vector is reevaluted by finding the maximum of all the possible notations. The result can be seen in Table 5.6.

components	A	B	C	D	E	F	G
transaction a	0	1	?	-	-	?	-
transaction d	0	0	0	0	-	-	0
diagnostic vector	0	0	0	0	-	?	0

Table 5.6: Result logic

The final vector changed so the iteration continues over the traces. The next trace is chosen to cover one remaining question mark at component F. Transaction c is suitable for this so it is chosen and added. The diagnostical vector is reevaluated. The result can be seen in Table 5.7.

components	A	B	C	D	E	F	G
previous vector	0	0	0	0	-	?	0
transaction c	0	-	-	-	-	1	?
diagnostic vector	0	0	0	0	-	1	0

Table 5.7: Result logic

The vector changed once again after evaluation so the iteration continues. However, when trying to choose a new trace the algorithm is looking for one to cover the number one mark in component F. There are no suitable traces like this. The vector does not contain any question marks either. This means that there are no suitable traces hence the algorithm stops and the result is the previously calculated diagnostic vector.

The result indicates that we were able to detect that the originally received error in component B wasn't the real error, but it was a result of some type of fault that started from component F.

Chapter 6

Summary

The report presents an approach to how distributed traces can be integrated into the methodology and processes of classical probing-based system diagnostics. It presented the extension of the standard algorithm by evaluating traces with a specific marking system that is able to increase the accuracy of the diagnostics results.

It makes the following statements represented by examples in each regarding section:

- **Traces from distributed tracing can be mapped into diagnostic probes.** The representation aligns with the classical representation of probes.
- **The minimum diagnostic probe set can be selected from the mapped diagnostic probes.** The selection algorithms used on probes are applicable to the transformed probes as well.
- **The minimum diagnostic probe set from distributed traces can be used for fault diagnosis.** The diagnostic sets are not only constructable but are functional too.
- **The fault and call graph information extracted from the traces can be used to increase the accuracy of single fault diagnostics.** Extra information can be gained from traces with their closer inspection that can help in the diagnostic process.
- **The fault and call graph information extracted from the traces can be used to increase the accuracy of single fault diagnostics even in the case of fault hiding.** This acquired extra information is able to help even if there is fault hiding in the system.

The plans for the future include the use of described methods and systems on a benchmark application (e.g., Train Ticket benchmark microservice system [2]). Furthermore, the result of the application of the methods in multi-fault scenarios should be investigated. The algorithm could be developed and sophisticated further to be able to handle these cases, too.

Bibliography

[1] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, 19:1–15, 2021.

[2] CodeWisdom Fudan University. Train Ticket: A Benchmark Microservice System (2021.). https://github.com/FudanSELab/train-ticket/.

[3] Jaeger UI. Jaeger UI. https://github.com/jaegertracing/jaeger-ui.

[4] Prometheus. Prometheus Project. https://prometheus.io/.

[5] Jesus Rios, Saurabh Jha, and Laura Shwartz. Localizing and explaining faults in microservices using distributed tracing. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 489–499. IEEE, 2022.

[6] Irina Rish, Mark Brodie, Sheng Ma, Natalia Odintsova, Alina Beygelzimer, Genady Grabarnik, and Karina Hernandez. Adaptive diagnosis in distributed systems. *IEEE Transactions on neural networks*, 16(5):1088–1109, 2005.

[7] Casey Rosenthal and Nora Jones. *Chaos engineering: system resiliency in practice.* O'Reilly Media, 2020.

[8] William R Simpson and John W Sheppard. *System test and diagnosis.* Springer Science & Business Media, 1994.

[9] The OpenTelemetry Project. OpenTelemetry Project. https://opentelemetry.io/.