Title
Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices

Author
Mathur, Mihir

Publication Date
2020

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Mihir Mathur

2020

ABSTRACT OF THE THESIS

Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices

by

Mihir Mathur
Master of Science in Computer Science
University of California, Los Angeles, 2020
Professor Miryung Kim, Co-Chair
Professor Ravi Arun Netravali, Co-Chair

Microservice architectures have gained prominence in recent years for building large-scale industrial distributed systems. However, microservice architectures make the usage of replay debugging, a powerful technique for finding root causes of faults, very challenging because of the polyglot (written in several languages) services, large accumulated state of services, and tight latency limits imposed by long hop-chains. This work attempts to provide a framework for enabling replay debugging in production microservice applications. We study 25 real-world faults in microservice systems collected from diverse sources, categorize these faults by fault symptoms, and create 15 application agnostic mutation operators for microservices. We then propose a language agnostic replay debugging framework for microservice applications that uses a distributed tracing system to record network requests and enables replay of those requests on cloned service containers running in a debug environment. A key component of this framework is an anomaly detector that uses span-level and container-level monitoring to detect fault symptoms found in our study and localizes faults to trace level so that faulty traces can be easily replayed to find the root cause. An open-source microservices application injected successively with the mutation operators is used for an evaluation that shows that our framework is upto an order of magnitude lighter-weight than language-specific recording tools such as Chrome DevTools or VisualVM and can help in finding root causes of 9 out of 15 mutations at a line or function level.

The thesis of Mihir Mathur is approved.

Harry Guoqing Xu

Ravi Arun Netravali, Committee Co-Chair

Miryung Kim, Committee Co-Chair

University of California, Los Angeles

2020

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to Prof. Miryung Kim for advising me during my undergraduate and graduate career at UCLA. I am thankful to Dr. Kim for giving me research opportunities, for several ideas used in this thesis, and for her constant words of encouragement. I am very thankful to my co-advisor, Prof. Ravi Netravali, for his support, ideas, and feedback over the past year which guided me throughout this project. I want to thank Prof. Harry Xu for several brainstorming sessions and for his inputs that helped me with this thesis.

I also want to thank other faculty and students of the UCLA Computer Science department who I've learned a lot from during my time as an undergraduate and graduate student. Finally, I want to thank my parents, Anup and Sangeeta Mathur, my brother, Raghav Mathur, and my girlfriend, Kyra Benowitz, for their emotional support, especially during the grim times of a global pandemic when I wrote most of this thesis.

CHAPTER 1

Introduction

In the past decade, industrial software systems have evolved significantly and one trend that has become increasingly common is large scale distributed systems being built with microservices architecture [2], a variant of the service-oriented architectural style. The core idea of the microservices architecture style is to manage the growing complexity of large systems by functionally decomposing these systems into a set of independent services and thereby unlocking easier scalability, maintainability, and faster iteration [7]. Major web and mobile software serving millions of requests per hour such as streaming services (Netflix and Spotify), ecommerce platforms (Amazon and Ebay) and ridesharing applications (Lyft and Uber) are built using microservices [18, 15]. Further, many more enterprises are starting or planning to switch to microservices from monolithic architectures because of the numerous advantages of this architectural style [2].

Despite the several benefits that the microservices architectural style provides, there remain critical challenges that make working with microservices harder than working with other traditional architectural styles such as layered or client-server architectures for building distributed systems. These challenges include debugging challenges such as fault localization [32] and performance debugging [12], operational challenges such as deployment, manual configurations, and team communication [9], and testing challenges such as systematic resiliency testing [14] etc. Given the widespread adoption of microservices in the industry as well as several challenges associated with this architecture, there is a need for research on tools and techniques for building and debugging microservice systems more effectively.

Among the challenges of working with microservices, overcoming debugging challenges is of critical importance for any production software since bugs can adversely affect end user experience and a company's revenue. Debugging is especially challenging for

microservices because of numerous reasons including:

1. **Large Accumulated State**: Accumulated state of services due to some production workload experienced can lead to several non-crashing errors such as timeouts/delays and high resource usages. Reproducing the accumulated state and corresponding workload in a debugging environment is complex which makes finding a root cause harder. Independent update and deployment of services adds to the complexity of reproduction of a system state since one would need to keep track of version changes of each service.

2. **Polyglot (i.e. written in several programming languages) System**: Many different languages and frameworks are used in a single microservices system. Therefore, language-specific replay debugging tools cannot be used for reproducing a system state. Fault localization is hard since disparate logs are generated due to which different sources need to be correlated for debugging non-crashing failures. Further, due to polyglot containerized services, lots of manual configurations (memory/cpu limits, restart policies, timeouts, circuit breakers, retries, what to log etc.) are needed for each service. Errors due to mis-configurations may happen during runtime and are hard to debug without reproducing the running system.

3. **Cascading Failures and Overheads due to Long Hop Chain**: A single request often triggers a long chain of requests to different services. Thus, failures can cascade: an observed error in one service might be due to bugs in other services; similarly a latent error in one service could propagate to several different parts of the system. The long hop chain also imposes a tight latency limit on each individual service since the latency that the end user experiences would be the sum of latencies of each service in the chain of requests. Consequently, any debugging solution that significantly increases latency would not be suitable for production systems.

Traditional debugging tools and techniques can be used to debug certain faults in microservice architectures. For instance, log analysis and breakpoint debugging can be used for finding the root cause if a failure is localized to a given service. However, for debugging non-crashing faults (such as timeouts, high latency or memory leaks), log analysis is

ineffective for pinpointing a root cause because: (1) it is hard to define, prior to program execution, what to record to find root causes of such faults and (2) causal relationship of logs across services cannot be examined easily. Breakpoint debuggers impose a single language constraint and cannot be used in production. Setting breakpoints on services running locally and running the system with same inputs does not guarantee that a fault will be reproduced since the local system state may be different from production. Another set of tools, distributed tracing frameworks such as Jaeger [10], Zipkin [26], and Dapper [29], track the causal and temporal relationships of service invocations for each request. These tools can help in localizing latency faults in microservices but have shortcomings: a large volume of traces is generated due to which a developer needs deep prior knowledge of the system to be able to query and get the trace that can lead to a latent fault. Further, even with a faulty trace, one cannot find the root cause of a fault by reproducing the fault scenario locally since the accumulated system state (which includes available resources to each service or the workload on each service) is not recorded within a distributed trace. **In aggregate, limitations of traditional techniques for debugging microservice application faults (discussed further in Section 2.1.2) are: (1) ineffectiveness in localizing the root causes of non-crashing failures at the log or trace level, (2) single language constraint, and (3) inability to automatically reproduce an entire system state in a local/debug environment**.

A potential solution that overcomes these limitations for finding latent bugs in a production microservices system is a framework that can proactively identify fault symptoms, mark traces corresponding to those faults, and enable easy reproduction of faults and symptoms in a debugging environment. Designing such a framework for production systems needs to address two orthogonal goals: (1) lightweight recording so that production systems can use the framework without excessive overheads and (2) finding the needle (i.e. the faulty trace) in a haystack (i.e. the set of all recorded traces) so that developers do not need deep prior knowledge of the system to identify which traces to replay.

For designing and evaluating a debugging framework that can meet these two goals, there is a need for a benchmark of faults based on real world fault and fault-symptoms prevalent in microservice applications. In this work, we analyze 25 microservices faults

from the previous literature, open-source repositories, and software Q&A website, Stackoverflow and categorize the faults by their symptoms. The analysis shows that a majority of these faults can be reproduced by capturing all network requests and responses, and replaying those requests on recorded service states. We derive 15 application-agnostic mutation operators from the faults that would show similar symptoms. We then design a record and replay framework (prototyped with Zipkin) that can reproduce failure symptoms for microservice faults on cloned service containers and can assist in finding root causes of faults. The proposed replay debugging framework is language-agnostic, incrementally integrable with production systems, and can help engineers to identify potentially faulty traces and then easily reproduce failure symptoms in a debugging environment.

The main contributions of this work can be summarized as follows:

- We give an analyses of deficiencies of traditional debugging techniques for debugging microservice faults (Chapter 2)
- We study 25 microservice faults, categorize them by symptom, and derive 15 application agnostic mutation operators from the collected faults that can be used to evaluate a microservices debugging framework on an arbitrary microservices application (Chapter 3)
- We propose a framework for record and replay debugging of microservice applications, defining the adequate information that needs to be recorded to replay executions for debugging common faults. As part of the framework, we give a mathematical formulation for using container-level (eg. memory %, cpu usage, etc.) or span-level (eg. latency) time series data for automatically detecting anomalous traces that are useful for replay debugging (Chapter 4)
- We evaluate our framework on an open-source microservices application and provide three case studies to show how our framework can be used for finding line-level or function-level root causes of latent faults. Nine out of fifteen of the mutations can be debugged using the methods described in the three case studies. Our framework imposes upto 10X less latency overhead in production than heavier language-specific recording tools such as Chrome DevTools or VirtualVM (Chapter 5)

CHAPTER 2

Background and Related Work

2.1 Background

Software architecture, analogous to building architecture, is a concretely defined set of design elements (including processing, data, and connecting elements) with a certain form; an *architectural style* (or pattern) is an abstraction of formal aspects from specific architectures that focuses on relationships and constraints of those design elements [27]. Architectural styles offer an outline of solutions to commonly occurring problems in engineering different kinds of software applications and these styles can be organized by application types. For instance, applications primarily concerned with shared memory could use a *blackboard* or *rule-based* architectural style, applications that require adaptable components could use a *microkernel* architectural style, applications that focus on messaging between sub-systems could use an *event-driven* or *publish-subscribe* architectural style, and applications that need to run on a distributed system could use a *client-server*, *broker*, *peer-to-peer*, *space-based* or *service-oriented* architectural styles [28]. These styles may not be mutually exclusive; several architectural styles could be combined to form a hybrid architecture for satisfying different application requirements. For example, if an application requires reacting to events while the logical processing runs over several physical machines connected over a network, it could be built using a hybrid architecture that combines the event-driven and the service-oriented architectural styles. As discussed in Chapter 1, one of the most popular architectural style in the industry is the microservice architectural style and it is used for building software as diverse as ridesharing mobile applications to e-commerce platforms. In the next subsection we discuss this style in more depth.

2.1.1 Microservices Architectural Style

The microservices architectural style is a variant of the service-oriented architectural (SOA) style, in which software is decomposed into processing elements (services) that communicate over a network. There are various definitions of the term *microservices* across the software engineering research literature. Fowler (2014) defines microservices as *an architectural style and approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API* [11]. Other definitions of microservice or microservices architectural styles include phrases such as "single functions", "autonomously developed", "bounded by contexts", and "fine-grained", "independent data stores". One of the key differences between SOA and microservices architecture is independent data stores in the latter. For example, if an e-commerce web application was built with an SOA, then services would be course grained such as commerce service and frontend service. However, if the same application was built with a with a microservices architecture, services would be fine-grained (the commerce service could be broken down into checkout service, cart service, payments service) and would have their own databases. For the purpose of this thesis, we will use Definition 1 when referring to the microservices architectural style.

Definition 1. ***Microservices architectural style****:*
Microservices architectural style is a method of decomposing a software system into small, independently developed and independently deployed processing components that work together, each with their own data components.

The popularity of this architectural style for building distributed systems is due to several advantages it provides over the more conventional pattern for building distributed systems, the monolithic client-server architecture. These advantages include agility of development and deployment, flexibility in changing constituent technologies, scalability of individual components, loose coupling between components, lower chance of regression, among other advantages. Table 2.1 lists some characteristics of the microservices architectural style versus a monolithic client-server architecture.

Property	Monolithic Client-Server Architectural Style	Microservices Architectural Style
	Example Application e-commerce application: ecommerce-demo[1]	Example Application e-commerce application: Online Boutique[2]
Functional Decomposition	Workload partitioned between two main components Eg. in ecommerce-demo: 'client' and 'server'	Numerous independent services handling separate business logic Eg. 10 independent services in Online Boutique: cartservice, currencyservice, checkoutservice etc.
Technology Stack	Locked in to a few languages and technologies. Eg. in ecommerce-demo: MEAN (MongoDB, Express, AngularJS, NodeJS) stack is fixed. Any new functionality must conform to this stack	Polyglot. Can support arbitrary number of technologies. Eg. in Online Boutique: cartservice (C#), currencyservice (NodeJS), checkoutservice (Go) etc. New services can be written in any language.
Code Organization /Version control & Code Visibility	Single repository contains entire source code. All developers have visibility into entire code base Eg. ecommerce-demo has a single repo	Usually each service has separate repositories, in which case developers may not have visibility into other services' source code. Though mono-repo, microservice applications also exist (Eg. Online Boutique is mono-repo)
Deployment & Scaling	Entire application is deployed together and scales together. Eg. if there was high load on a particular feature of ecommerce-demo, the entire application (i.e. the client and the server would scale together and run on multiple hosts.	Services are deployed and scaled independently from one another. Eg. checkoutservice of Online Boutique could scale to several replicas running on multiple hosts, without any other service scaling.
Testability	Writing and running integration and end-to-end tests is easier. Eg. the entire ecommerce-demo application can be run and tested easily on a single host.	Harder integration and end-to-end testing. Eg. All services of Online Boutique would need to run (potentially on multiple hosts) in compatible versions for any end-to-end test

Table 2.1: Characteristics of Microservices Architectural Style versus Monolithic Client-Server Architectural Style

While the microservices architectural style provides several benefits for building distributed systems, it also brings several disadvantages. One of the major disadvantages is that debugging becomes much harder for systems built with microservices compared to systems built using other common architectural styles. The next section discusses deficiencies of traditional debugging techniques for microservices.

2.1.2 Traditional Techniques for Debugging Microservice Faults

While there exist several debugging techniques for finding and fixing faults of programs, most of those techniques are optimized for programs running on single machines. In this section, we examine three traditional debugging techniques in the context of debugging microservice applications: log analysis, breakpoint debugging, and distributed tracing .

[1]https://github.com/ratracegrad/ecommerce-demo

[2]https://github.com/GoogleCloudPlatform/microservices-demo

2.1.2.1 Log Analysis

In this debugging technique, records generated during program execution are analyzed to find the root-cause of a failure. There is often a much higher volume of logs generated in microservice architectures than monolithic client-server architectures because of the higher number of network calls and steps in a transaction [21], which increases the search space for log lines that can help in debugging a failure. Moreover, log analysis is in general not well suited for debugging non-crashing failures like high latencies or memory leaks because it is hard to define, prior to program execution, what to record to find root-causes of such faults. Further, it is hard to track the causal relationship of logs across services, which is often required for debugging such faults. As discussed in Chapter 3, non-crashing faults such as delays/timeouts and high resource usages are two commonly occurring categories of microservices faults, and the ineffectiveness of log analysis to pinpoint the root-cause of such faults from a given set of logs makes it a weak debugging technique for microservices.

2.1.2.2 Breakpoint Debugging

Using debuggers (such as GDB[3] and PDB[4]) or IDEs (such as Visual Studio[5] and Eclipse[6]) developers can set breakpoints at arbitrary points in a program and pause execution to examine the program state. While breakpoint debugging can be powerful in debugging single process programs, setting breakpoints across different services and examining program state is very hard because of two reasons: (1) the services are written in many different programming languages and run in containerized environments (2) the program state is much more complex than that of a single process since the state of the distributed system involves the services running on different machines, the available resources to each service, the number of replicas of each service, the workload on each service etc. So setting breakpoints successfully might not lead to a root-cause of a bug in a microservices

[3]https://www.gnu.org/software/gdb/

[4]https://docs.python.org/3/library/pdb.html

[5]https://visualstudio.microsoft.com/

[6]https://www.eclipse.org/ide/

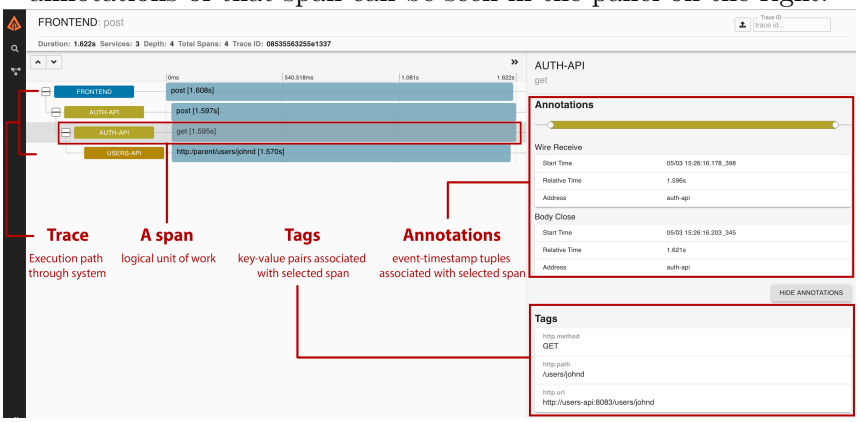
architecture if the state and workload is not re-created, which cannot be done in an automated way using single-language debuggers or IDEs. Further, a developer needs to guess locations for setting breakpoints, which can be hard in a microservices architecture since developers often do not have the knowledge (or in some cases access) of all potentially faulty services.

2.1.2.3 Distributed Tracing

Distributed tracing is a widely used technique for tracking the sequence of operations performed by different components in a distributed system. The two key terms of distributed tracing are: *span*, "a logical unit of work that has an operation name, the start time of the operation, and the duration" and a *trace*, "a data/execution path through the system that can be thought of as a directed acyclic graph of spans" [10]. An arbitrary number of spans can be created within a service which allows for tracing to be done at any granularity: at the service level, at the function level or even at the line level. Distributed tracing frameworks like Zipkin or Jaegar propagate context information for each request and can record the causal relationships, latency, responses, and failure rates of each service. Figure 2.1 demonstrates an example trace and its spans as they appear in Zipkin's web user interface.

Figure 2.1: Zipkin Web UI

An example trace and its associated spans are shown. A span is selected and the tags and the annotations of that span can be seen in the panel on the right.



While very powerful, distributed tracing for debugging microservices has some challenges. First, instrumentation of services has to be done a priori of program execution

by multiple people (i.e. developers of different services) and it is not always clear what is the minimum yet sufficient instrumentation for being able to debug future errors. Second, a large volume of traces are produced in a microservices architecture if all functions in each service are instrumented and finding a fault from traces becomes a needle in a haystack problem that requires experts for debugging. Third, a failure cannot be easily reproduced using Zipkin or Jaegar because an accumulated state of services might be unknown (if not enough information was recorded) or if services have evolved since the trace was produced.

Therefore, traditional debugging techniques such as log analysis, breakpoint debugging, and distributed tracing for debugging distributed systems such as microservice applications have deficiencies including ineffectiveness in pinpointing root-causes of non-crashing failures, single-language constraint, lack of automation, and inability to reproduce system state.

2.2 Related Work

2.2.1 Record and Replay Debugging

Record and replay debugging is a technique in which certain information is logged during program execution in order to replay the program again in a debugging environment to diagnose bugs. A major challenge of designing a record and replay system for distributed systems is the very large state space consisting of memory and configuration of all the physical machines, the state of the network, the volume of concurrent requests (load) etc. One of the first research projects for replay debugging of large distributed applications was Liblog [13], a tool that provides a shared library to services for logging and uses Lamport clocks on all messages between services for keeping track of execution order. While Liblog can replay program executions faithfully, its two main drawbacks are: (1) the logging library only works with C/C++ and supports only POSIX applications (2) it introduces significant runtime overheads such as 18% throughput reduction. Several other tools for replay debugging [17, 30, 25, 20, 3] work with only one programming language. These tools would not be able to replay executions across services written in different programming languages, which is very typical in microservice architectures.

The framework proposed in our work is programming language agnostic.

The closest related work to our work is Parikshan [1], a production bug replaying framework for service-oriented applications. The key insights of this paper are that the state of service oriented applications necessary for reproducing bugs can be captured by replicating each buggy production container and that bugs can be faithfully replayed by sending same network traffic of production containers to replicas of buggy containers. Traces for a faithful replay of bugs are never recorded, instead, live network traffic is forwarded to replica debug containers. Parikshan lets developers recreate a production system state in sandbox environments by relaying network packets to replicas of suspect services.

While Parikshan is an effective framework for *on-the-fly* or live debugging of services, it cannot be used for debugging errors that happened before container replicas were created. For example, if a developer is notified that one hour ago some users were experiencing very slow responses, that developer would not be able to debug this fault since there is no recording of the system state or the inputs that lead to the fault since the problem occurred before the replica containers were created. In such a case, the developer would create replica containers and wait for that fault to happen again. Faults 3,9, and 25 in Appendix A are examples of faults that are triggered by a particular user input and do not persist. Not recording any trace implies that a developer has to debug the error exactly when it is happening i.e. a developer has to find a root cause fast because the fault could stop persisting and there is no way to go back to the faulty state. The authors justify not recording any trace by noting that capturing sufficient traces to faithfully replay bugs in a debugging environment can result in performance overheads balooning up to 2-10x, which is unacceptable for use in production SOA. However, they ignore the fact that companies with production microservice systems already have some level of distributed tracing [19] that can be used without adding excessive overheads. Our framework leverages the existing distributed tracing present in most production microservices systems [19] to record certain information (mentioned in Section 4.2) that enables a developer to debug faults that were observed before the cloned containers were created.

2.2.2 Anomaly Detection

Anomaly detection, the problem of finding patterns in data that do not conform to expected behavior, has been researched across several application domains which has led to several techniques for automatic identification of past anomalies or prediction of future anomalies [4]. Such anomaly detection techniques have been used in the context of microservice application monitoring and debugging. We discuss three such systems: Seer [12], MEPFL [33], and ADaaS [23].

2.2.2.1 Seer

Seer [12] is a performance debugging system for microservices that predicts upcoming performance violations such as high tail latency or low throughput in different services, using anomaly detection. For predicting such upcoming violations, deep neural networks trained on historical execution traces annotated with violations are used to identify the faulty microservice. Once a potentially faulty microservice is identified, Seer uses per-node, low-level hardware monitoring primitives such as performance counters to find the root cause of the violation and to recommend steps on how to prevent the performance degradation. While Seer is able to detect and avoid a majority of imminent performance violations, the authors noted that "violations that were not avoided correspond to application level bugs, which cannot be easily corrected online". Our framework can help in fixing application level bugs (such as a memory leak when a particular execution path is taken) since it enables a developer to reproduce a system state in a debug environment by replaying anomalous traces (discussed in Section 4.3).

2.2.2.2 MEPFL

MEPFL (Microservice Error Prediction and Fault Localization) [33] is a system that uses anomaly detection models such as Random Forests and Multi Layer Perceptron trained on traces obtained from execution of fault-injected version of services to predict latent errors in microservice applications. While MEPFL is able to localize the error to the service-level with high accuracy, the debugging process still mandates additional effort from an expert to precisely find the root-cause at the line level. The framework presented

in our work provides a simpler method for a developer to localize a fault to the line level for certain fault types such as resource leaks or high latency once an anomaly is detected.

2.2.2.3 ADaaS

Another related system is ADaaS (Anomaly Detection As-a-Service) [23]. ADaaS provides an architecture for combining different statistical anomaly detection modules (such as a mean shift anomaly detector) to seamlessly monitor complex cloud systems. While ADaaS provides a declarative method for controlling anomaly detection logic, it does not provide any actionable insights to fix the anomalies.

Anomaly detection models from Seer, MEPFL, and ADaaS can be integrated in the Anomaly Detection component of our framework (described in Section 4.4). For prototyping our framework, we used the lightweight statistical anomaly detection modules outlined in ADaaS.

2.2.3 Containerization and Container Cloning

Containerization is the technique of encapsulating some source code and all of its dependencies and associated configurations into images that can be run reliably across all computing environments. Unlike virtualization, containerization is a lightweight technique because it uses a host machine's operating system instead of bundling a copy of the operating system like Virtual Machines do [8]. Using containerization for microservice application development is the current industry standard. While there exist techniques [24, 22] for live cloning containers, Parikshan [1] is the only tool we know that leverages container cloning for debugging microservice applications. In Parikshan, a developer has to specify which containers to clone and when, whereas in our proposed framework containers can be either cloned on a schedule or containers to be cloned can be automatically inferred from faulty traces (described in Section 4.2.3).

CHAPTER 3

A Benchmark of Microservice Applications Faults

There have been two prior works on microservices debugging that categorize microservice faults: X. Zhou et al. [31] organize faults collected from an industrial survey by their root causes and the maturity levels of debugging (log analysis, visual log analysis, or visual trace analysis) and Nipun Arora et al. [1] organize faults collected from issue trackers of open-source SOA applications into four categories (performance, semantic, concurrency and resource leaks). However, these categorizations have limitations: 1) both of these prior works study faults from mutually exclusive sources, and 2) these works do not categorize by fault symptoms. Understanding fault symptoms is important since the observation of a symptom is often how the debugging process starts. Further, fault symptoms can help with the automatic identification of which traces to replay. To overcome these limitations of existing microservice fault categorizations, we conduct our own study. We answer the following research question in this chapter:

What are common faults in microservices architectures and how can these faults be categorized by symptoms?

3.1 Methodology

Microservices faults from three different sources were collected:

1. **Literature:** for obtaining microservices faults from the published literature, search terms "microservices debugging" and "soa debugging" were used on Google Scholar in January 2020. We found six relevant papers [1, 31, 16, 5, 12, 32] and each paper was skimmed for descriptions of faults. Faults relevant to microservices were recorded along with more context and individual root causes in a spreadsheet.

2. **Github Issues:** Issues of 3 open-source industrial microservice projects (Site-

Where[1], Open-Loyalty[2], Gizmo[3]) were examined in January 2020. We found these three repositories listed under the Industrial Projects section of a microservices project list[4]. Issues were filtered by the "bug" label. Gizmo and Open-Loyalty had 4 closed issues with a "bug" label. Sitewhere had over 100 issues with the "bug" label, so we used search term "service" to reduce the number of issues to about 20. Each of these issues was manually examined. Bugs that were due to the microservices architecture of the project and had a bug-fix mentioned were recorded in the spreadsheet along with more context and root causes.

3. **Stackoverflow:** Search terms "microservice bug", "microservice fault", "microservice debug", "microservice error" were used on Stackoverflow search in January 2020. Results were sorted by relevance. Each search result on the first page (15 results) was manually examined. If the question was about a microservices bug and there was a fix mentioned in the answers, then the question was recorded in the spreadsheet along with more context and root-cause.

There were two constraints while collecting faults from the aforementioned sources: 1) faults whose symptom and root-cause were within one service were ignored since these faults are not due to the microservices architecture of the system but rather an internal error within a service, and 2) we set a limit of six faults from any one source to allow for a diversity of sources i.e. no more that six faults (which is <25% of total faults) are from the same research paper or the same open-source repository.

25 total faults were collected by this methodology. The details of each fault are in Appendix A.

[1]https://github.com/sitewhere/sitewhere

[2]https://github.com/DivanteLtd/open-loyalty

[3]https://github.com/nytimes/gizmo

[4]https://github.com/davidetaibi/Microservices_Project_List

Fault Symptom: Errors and Exceptions		
Fault	**Root Cause**	**Source**
State change updates sending after shutdown and throwing exceptions	When shutting down an instance, state updates are still being sent after the heartbeat service is terminated since the updates thread is not terminated	https://github.com/sitewhere /sitewhere/issues/726
Two 502 responses triggered upon attempting the deletion of the telephone accounts	Error in interaction with Cassandra.	Microservices Monitoring with Event Logs and Black Box Execution Tracing (Pg 4). https://ieeexplore.ieee.org/document/8826375
Zuul Forwarding error - Internal Server error 500	Root-Cause: Docker and Kubernetes configuration: Zuul path not setup properly in yaml file	https://stackoverflow.com/questions/ 55026430/zuul-forwarding-error-internal-server-error-500
com.netflix.discovery.shared. transport.TransportException: Cannot execute request on any known server	Configuration YAML did not have 'register-with-eureka' flag set to false	https://stackoverflow.com/questions /46131196/com-netflix-discovery-shared-transport-transportexception-cannot-execute-reques
The payment service of the system fails	Root-Cause: The overload of requests to a third-party service leads to denial of service	Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. TSE'18

Table 3.1: Microservices Faults with Error and Exceptions observed

3.2 Fault Organization by Symptom

For categorization of faults, the first 10 faults were analyzed and some candidate categories of symptoms were created. Then, each fault was put in a category or a new category was created if none of the existing categories seemed to match the fault.

Our study of microservices faults showed that for fixing a microservices fault, developers observe one of 4 symptom categories: (1) Errors/Exceptions (2) Delays/Timeouts (3) High Resource Usage, and (4) Unexpected Output. The next section discusses each category and provides example faults.

1. **Faults identified by Errors/Exceptions** These are microservices faults where the debugging process was started by the observation of an error or an exception (such as a HTTP 500 Internal Server Error). Table 3.1 gives some example faults whose debugging started with an observation of Errors/Exceptions. The total number of faults in this category was nine and Appendix A.1 gives more detailed descriptions of each of those faults.

2. **Faults identified by Delays/Timeouts** These are microservices faults for which the debugging process was started when an unexpected delay was observed in getting a response or if there was a timeout observed for a request. Table 3.2 gives

Fault Symptom: Timeouts and Delays		
Fault	**Root Cause**	**Source**
Redis: connection with the slave times out and it's unable to sync because of the large data	a lower output buffer limit.	Replay without Recording of Production Bugs for Service Oriented Applications. ASE'18
some of the user requests which were dealing with complex scripts (Chinese, Japanese), were running significantly slower than others.	Bug caused due to multiple calls in a loop	Replay without Recording of Production Bugs for Service Oriented Applications. ASE'18
a timeout of the server –code 504– occurred (reported in client logs)	Connection refused by cassandra (call in 3rd degree service), which is used to store authentication credentials and profile information in Clearwater.	Microservices Monitoring with Event Logs and Black Box Execution Tracing (Pg 4). https://ieeexplore.ieee.org/document/8826375
AppScale datastore service is slow to respond	Injected fault to system for showing root cause analysis system works. This fault injection logic activates once every hour, and slows down all datastore invocations by 45ms over a period of 3 minutes	Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications. WWW'17
A service is slowing down and returns error finally	Endless recursive requests of a microservice are caused by SQL errors of another dependent microservice	Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. TSE'18
MongoDB server selection timeout exceeded	If the microservices start before a MongoDB replica set has time to initialize, there are cases where the server s election timeout (30s) is exceeded.	https://github.com/sitewhere/sitewhere/issues/721

Table 3.2: Microservices Faults with Delay or Timeout observed

some example of such faults. The total number of faults in this category was seven and Appendix A.2 gives more detailed descriptions of each of those faults.

3. **Faults identified by High Resource Usage** These are microservices faults for which the debugging process was started when some resource used by a service such as the number of CPUs or memory was anomalously high. Table 3.3 gives some examples of such faults. The total number of faults in this category was four and Appendix A.3 gives more detailed descriptions of each of those faults.

Fault Symptom:,High Resource Usage		
Fault	**Root-Cause**	**Source**
Loading invocation by unique id in InfluxDB can overload RAM	Due to the way that command invocations are indexed in InfluxDB, getting one by unique id can result in loading all event data in RAM.	https://github.com/sitewhere/sitewhere/issues/655
Unusual memory usage in the Glassfish application server, causing error logs to be generated in the Nginx web server	Persistent memory leaks in a container.	Replay without Recording of Production Bugs for Service Oriented Applications. ASE'18
Memory leak observed in a specific version	"Gizmo's server.Router implementation uses gorilla/context under the hood to allow httprouter to pass parameters. This leads to a massive memory leak when running with 1.7 as the gorilla/context never gets cleared."	https://github.com/nytimes/gizmo/issues/74

Table 3.3: Microservices Faults with High Resource Usage observed

Fault Symptom: Unexpected Output observed		
Fault	**Root Cause**	**Source**
Messages are displayed in wrong order	Asynchronous message delivery lacks sequence control	Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. TSE'18
The number of parts of a specific type in a bill of material (BOM) is wrong	An API used in a special case of BOM updating returns unexpected output	Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. TSE'18
A default selection on the web page is changed unexpectedly	The key in the request of one microservice is not passed to its dependent microservice	Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. TSE'18
Missing values when API call made to endpoint	missing argument in route of service endpoint. By changing api/level to api/level?perPage=total_level, this bug can be fixed	https://github.com/DivanteLtd/open-loyalty/issues/78

Table 3.4: Microservices Faults with Unexpected Output observed

4. **Faults identified by Unexpected Output** These are microservices faults for which the debugging process was started when an unexpected output from a service was observed. Table 3.4 gives some examples of such faults. The total number of faults in this category was five and Appendix A.4 gives more detailed descriptions of each of those faults.

3.3 Mutation Operators for Microservices

For creating a language agnostic debugging framework for microservices, we needed a benchmark of mutations representative of real-world faults. We created application-agnostic mutation operators for microservice symptoms that could show similar symptoms to faults collected in the study. One goal for creating these operators was to keep the code changes needed for the mutation as minimal as possible, while the mutations show symptoms similar to real-world faults so that other developers can easily use these operations within their applications for testing debugging tools. These mutation operators can be categorized into Network Mutations, Configuration Mutations, Resource Mutations and Response mutations. Table 3.5 lists the mutation categories and provides code diff examples in Python and configuration file diff examples in .yaml.

Mutation Type	ID	Mutation Name	Mutation Description	Example Mutation	Deterministic Fault?	Symptom Similar To
Network Mutations	1	Consistent Delay	Responding service sleeps for some fixed duration before sending response	Python: + time.sleep(.2)	Yes	F12
	2	Random Delay	Responding service sleeps for some random duration before sending response	Python: + time.sleep(random.randint(0, 10))	No	F14, F15
	3	Abort/Throw Exception	Responding service aborts request or throws an exception.	Python (Flask): + abort(404)	Yes	F16, F1, F2
	4	Timeout	Responding service does not respond and lets request timeout	Python: + while True: pass	Yes	F13, F11, F10
Configuration Mutations	5	Environment Variable Edit	Some env variable line is edited in configuration YAML	.yaml: - REDIS_PORT:6379 + REDIS_PORT: 6380	Yes	F7
	6	Flag Toggle	Some configuration boolean of container is inverted	.yaml: - register-with-eureka:true + register-with-eureka:false	Yes	F5, F6, F8
	7	Edit Restart Policy or Timeout	Modify restart policy of container or service	.yaml: + restartPolicy: Never	No	F11
	8	Replica Reduction	Reduce number of replicas for a deployment	.yaml: - replicas: 3 + replicas: 1	No	F4
Resource Mutations	9	Memory Leak	Some execution path in responding service leaks memory	Python: + while len(x) < 10: + x += bytearray(256000000)	Yes	F19, F18, F17
	10	Container memory reduction	Reduce available memory to container	.yaml: - memory: 50M + memory: 5M	No	F20
	11	High CPU	Some execution path in responding service is CPU intensive	Python: + while(i < 100000): + i++	Yes	F4
	12	Container CPU reduction	Reduce number of available CPUs of some container	.yaml - cpu: '0.25' + cpu: '0.025'	No	F4
Response Mutations	13	Incomplete Response	Remove field from response of responding service	Python: - return (a,b) + return (a)	Yes	F23, F25
	14	Out of order async reception	Remove sequence control logic in requesting service	Python: - r = await client.get('example.com/') + r = client.get('example.com/')	No	F21, F22, F24
	15	Response Type/Field Mutation	Change types in response or remove field	Python: - return x : 5 + return x: "5"	Yes	F9, F3

Table 3.5: Mutation Operators for Microservice Applications

The example mutation column gives code samples in Python or .yaml. + indicates lines added and - indicates lines removed for introducing the mutation. The last column indicates which faults mentioned in Appendix A have symptoms similar to the mutation operator of that row.

CHAPTER 4

Record and Replay Framework for Microservices

The framework proposed in this thesis attempts to improve upon existing debugging methods for microservices, especially for debugging latent or non-crashing faults. At a high level, the proposed framework:

- Leverages distributed tracing for language-agnostic recording and prescribes what and how to record
- Creates a debug environment with cloned containers and lets a developer replay traces on those containers to reproduce fault symptoms and enables them to find the root cause.
- Encourages *preemptive* debugging by detecting anomalies and forwarding potentially faulty traces to the developer.

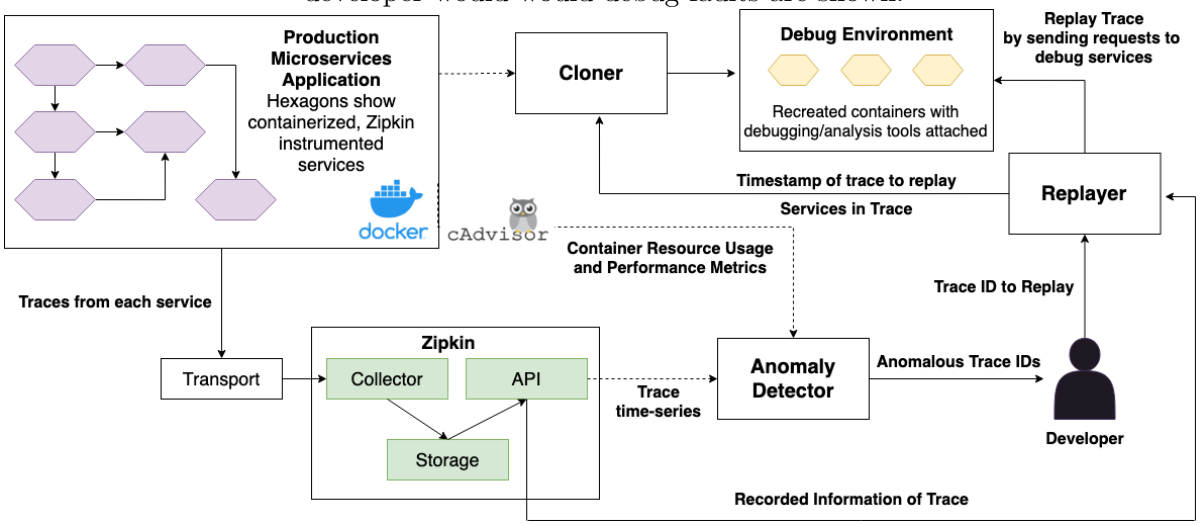
This chapter describes the design objectives and implementation details of the proposed record and replay framework for microservices. Figure 4.1 illustrates the setup of the components of the framework.

4.1 Design Objectives

The main goal of the proposed debugging framework is to reproduce faults of production microservice systems in debug environments and assist in finding root cause. For making this framework more useful for debugging production systems than simple distributed tracing or monitoring, two orthogonal goals should also be addressed: 1) lightweight recording and replay so that production systems can use the framework without excessive overheads and 2) finding the needle (i.e. the faulty trace) in a haystack (i.e. the set of

Figure 4.1: Setup of the Debugging Framework for Microservices

Cloner, Replayer, and Anomaly Detector are the three main components of this framework. The interactions of the production application, these components, and the developer would would debug faults are shown.



all recorded traces) so that developers do not need deep prior knowledge of the system to identify which traces to replay.

The following design objectives were derived from these goals:

1. **Fault Reproduction:** the framework should be capable of reproducing common microservices faults (listed in Table 3.5) in debug environments for record and replay debugging.

2. **Language Agnostic:** Since one of the major benefits of the microservices architecture is the polyglot nature of services, a good replay debugging framework should work irrespective of the languages chosen by developers of different services.

3. **Easy Integration with Production Systems:** Any effective debugging framework should have the ability to easily integrate with a system. We want our framework to be incrementally integrable with existing microservice applications without major architectural overhauls or development overheads.

4. **Low Overheads:** We envision recording to be always-on because several microservices bugs are latent. Therefore, the framework should favor low recording overhead over perfectly faithful replay so that users are not impacted and costs of always-on

recording are not significant. We argue that common microservices faults can be replayed without recording the entire system state.

5. **Automatic Actionable Insights:** One of the drawbacks of popular distributed tracing frameworks is that developers need some insight about a fault to successfully query and retrieve relevant traces. A framework that provides actionable insights by learning from historical traces and system data patterns can be very effective for debugging latent faults.

The next two sections describe how to record and replay executions in a microservices application while accomplishing the aforementioned design objectives.

4.2 Record

4.2.1 Instrumenting Services

As per the design objectives, the recording technique should not only be language agnostic, but should also be able to be easily integrated with existing production systems without significant overheads. Distributed tracing tools would work well for recording since these tools provide APIs for all major programming languages and introduce low performance overheads. Moreover, most companies that have large scale distributed systems already have end-to-end distributed tracing tools integrated with their systems. A recent analysis of systems of major internet companies including Google, Netflix, Facebook, Yelp and Uber found that frameworks such as Zipkin, Dapper, Jaeger, Brave, Zalando are used for end-to-end distributed tracing [19]. The most commonly used tracing framework was Zipkin, used by 14 of the 26 companies surveyed [19]. Therefore, Zipkin was used for prototyping our framework for recording service level traces.

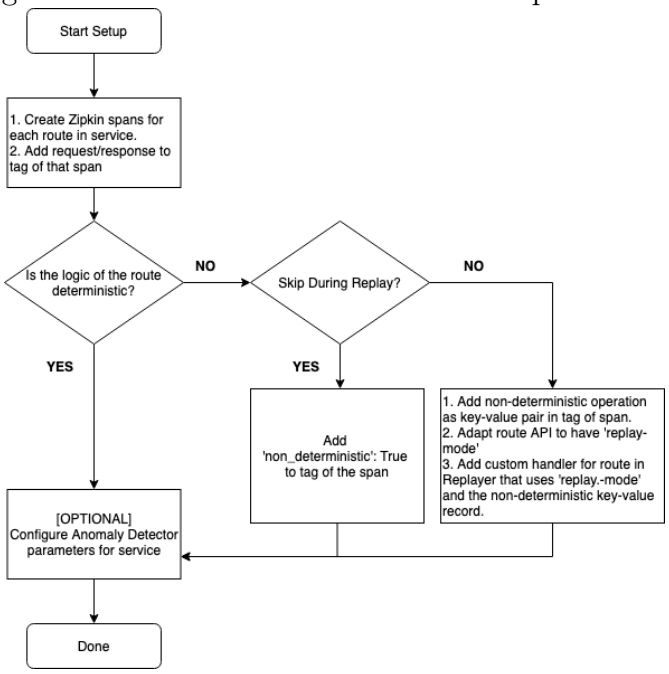
Zipkin is a lightweight distributed tracing system that creates a unique correlation ID for each request received by a microservices application and propagates this ID to all services in the invocation chain of the request using HTTP headers. Microsecond-precision timing events are recorded as the request goes through the call chain of services, enabling Zipkin to track latency of each service. Services written in major programming languages such as Java, Python, JavaScript, Go, C++, among others can be easily

instrumented with Zipkin using pre-existing libraries. Zipkin not only enables RPC level tracing, but also provides a lot of flexibility in what to record since spans can be created at the granularity of line level. Further, more information can be recorded within each span using *tags* (key-value pairs) and *annotations* (timestamp-value pairs). A sample Zipkin span is showed in Figure 4.5. Zipkin has 4 main components: (1) a *collector* daemon that collects trace data sent over HTTP by services and that validates, stores and indexes the data (2) a *storage* component for storing trace data (3) a *query service* that provides a simple JSON API for retrieving trace data, and a (4) a *web UI* (Figure 2.1) for visualizing a dependency graph of services and the information recorded within each trace on a timeline [26].

4.2.2 What to Record?

In an ideal scenario, we would record minimum yet sufficient data for faithfully reproducing system state from the given recording. However, it is hard to prove what data is sufficient to capture the state of an arbitrary microservices system. Even if we could define a set of elements that should be recorded for perfectly faithful replay, as noted in [1], recording sufficient information for faithful replay can lead to upto 10X performance overhead which is unacceptable in production microservice systems.

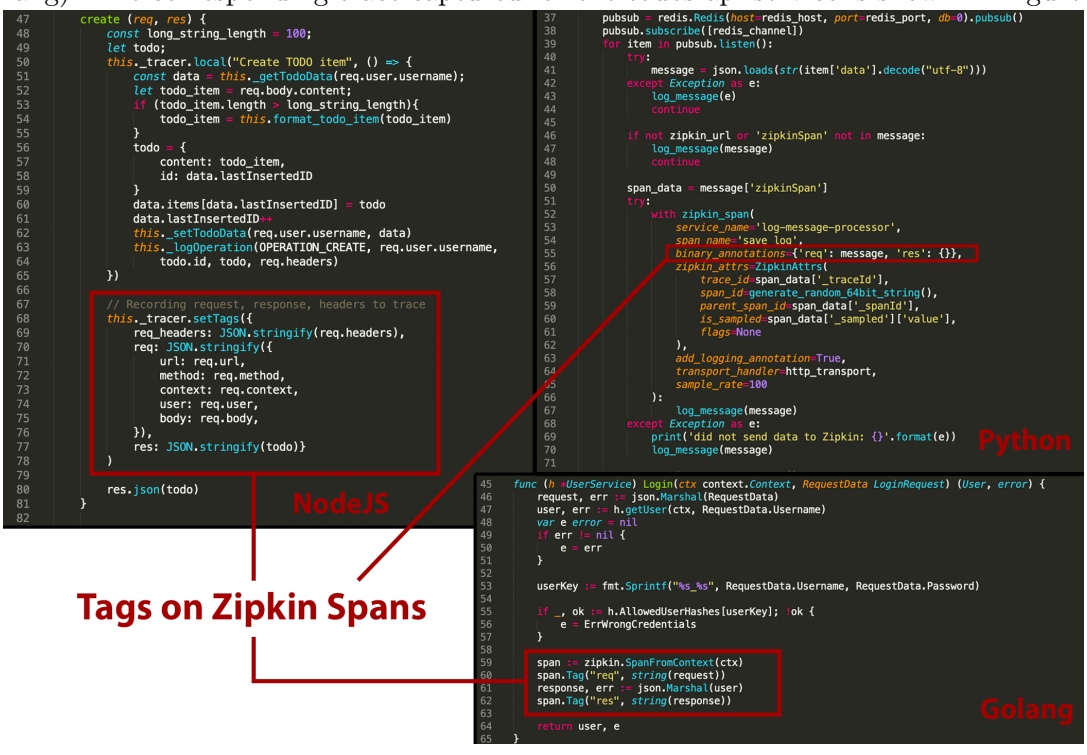
Figure 4.2: Service Instrumentation Setup Flowchart



What matters more for a debugging framework than perfectly faithful replay is whether common faults and fault symptoms can be reproduced. The fault analyses from Chapter 3 shows that several microservices faults can be reproduced by replaying network requests on each service. Therefore, a good starting point for recording data is capturing network requests at the service level. For example, if services are communicating over HTTP/2, then for each request the following should be recorded: Request method (HTTP GET/POST) and headers, request URL, request payload, response payload. This data can be recorded as tags (where the key is `req` and `res` and values are request and response data), on Zipkin spans by instrumenting each service at the route level. Figure 4.2 gives a flowchart for a service developer to instrument their service. Examples of tag instrumentation on Zipkin spans of three routes of the benchmark application used in this work (microservices-app-example[1]) is shown in Figure 4.3.

Figure 4.3: Tag Instrumentation in Different Services

Tags can be recorded on Zipkin spans using libraries for different programming languages. The three example code snippets shown here are from the benchmark application's instrumented services: todos-api (NodeJS), log-message-processor (Python), and auth-api (Golang). The corresponding trace captured for the todos-api service is shown in Figure 4.5



[1]https://github.com/elgris/microservice-app-example

Recorded request and response data could be sufficient for re-creating system state if services are deterministic. However, there exist several sources of non-determinism in a distributed system which means that the same set of inputs to a program running on a distributed system can result in different behaviours in terms of output, response time or side effects. While perfectly capturing all non-determinism is very challenging and imposes massive overheads, several sources of non-determinism in a distributed system could be captured in a lightweight way. The next sub section enumerates sources of non-determinism in microservices architectures and describes how they are captured (or how they are not) in our framework.

4.2.2.1 Sources of Non-determinism in microservices architectures

1. **Resource Non-determinism** In a microservices architecture, resources such as CPU, memory and network are allocated dynamically to services depending on workload and are a source of non-determinism. For instance, a request could fail if services do not have adequate resources at the time of request, but the same request could succeed if adequate resources are allocated. Since services are typically containerized using a platform like Docker, a time series of resource allocation and utilization can be recorded by polling containers at regular time intervals using a lightweight monitoring tool such as cAdvisor[2].

2. **Service Non-determinism** Services themselves can have non-deterministic behavior if the program logic is based on internal state that depends on a container's file changes, memory or databases. Some of this non-determinism can be captured by cloning containers (discussed in Section 4.2.3). Further, services could use random number generation to perform non-deterministic operations. For services doing such operations, a key-value pair of variable and generated random value can be recorded in the tag of a Zipkin span. Services could also have multi-threaded functions and output could depend on thread execution order. Callers of such endpoints or functions should set a 'non-deterministic operation' flag to true on the tag of the parent span.

[2]https://github.com/google/cadvisor

3. **Network Non-determinism** The rate of data transfer over a given path of nodes is non-deterministic. For the same request made at different times to a service that calls several other services, the response time might be different due to the varying network bandwidth. Network non-determinism can affect the service latency, which is a critical metric in microservice systems. Network IO rates can also be recorded using cAdvisor.

4. **Asynchronous Request Non-determinism** In microservices architecture, one request to a service can fan-out several asynchronous requests to other services. Due to network non-determinism, the order of responses received can be different. However, with sequence control logic in services, non-determinism due to asynchronous requests can be avoided.

5. **Non-Determinism from third parties** Services could make calls to external APIs for performing some operation which could be a source of non-determinism. This type of non-determinism cannot be handled by our framework. Callers of such external APIs should set 'non_deterministic' flag to true on the tag of the parent span so that the replaying system can avoid making an external request.

4.2.3 Container Cloning

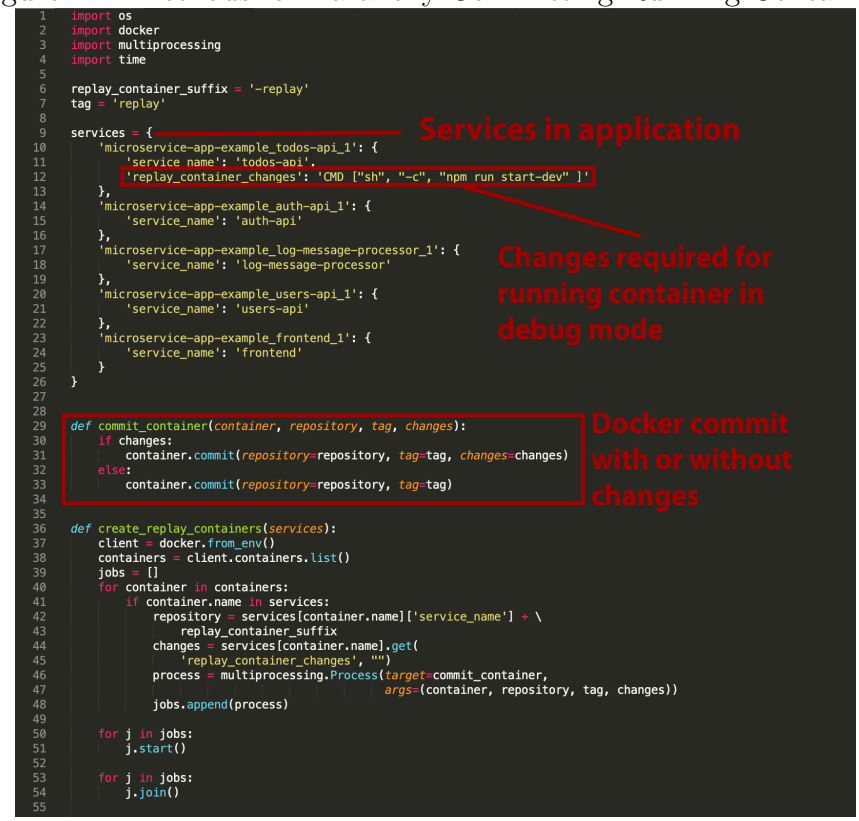
Containers solve the problem of running a software across different computing environments reliably. The Docker platform is the industry standard for running containerized services [6]. For recording the state of a running Docker-containerized service, our framework has an independent service, Cloner, whose job is to record running containers and manage the creation of debug containers with attached debugging tools in a separate environment for a developer. The Cloner can be configured to create a debug image for each service every time that service's production container is restarted or at any arbitrary point of time.

There are two alternatives for the underlying technique that the Cloner can use: (1) Checkpoint/Restore In Userspace (CRIU)[3] and (2) Committing the changes of the container to a new image. CRIU can save the snapshot of a running container on disk

[3]https://www.criu.org/

and then restore and run the container in exactly the same state as it was in during the snapshot, whereas committing a container can save all file changes in a new image that can be rerun as a separate container. While CRIU can effectively live migrate running containers, it is an experimental feature in Docker and neither has a robust API nor is fully supported on all operating systems. Committing a container using Docker Commit[4], is much more stable and more widely used. Committing a running container to an image can also be used to recreate the same container state if we send the same sequence of requests (as received by the originally running container) to the recreated container, while keeping a track of non-determinism. Since we record all requests to each container using Zipkin, we can recreate the container state on a debug container. Docker commit provides another benefit: the debug container can be configured to run in debugging mode. So whenever a developer wants to replay a trace, all debug containers could be run with the developer's preferred tools automatically attached.

Figure 4.4: Methods for Parallelly Committing Running Containers



There are two challenges of cloning microservice containers: (1) minimize the time that the containers are paused for least service disruption and (2) cloning containers in

[4]https://docs.docker.com/engine/reference/commandline/commit/

a consistent state i.e. the times at which different containers are copied should be as close as possible. Docker Commit can tackle the first challenge since it can create a copy of a container within a few milliseconds. For addressing the second challenge, separate parallel processes that start at the same time could be created for cloning each container. Figure 4.4 shows a function '`create_replay_containers`' that takes in as input the set of containers that need to be cloned and starts parallel Docker Commit processes for creating replay containers.

The prototype of Cloner saves the committed images on the physical machines running production services. However, it is easy to upload images to container registries such as Google Container Registry[5] or Docker Hub[6] and download those images to a developers local machine.

4.3 Replay

Replaying the captured traces requires creation of debug service containers, reconstruction of requests from Zipkin traces, and emission of those requests to services at time intervals consistent with original requests. These tasks are performed by the Replayer service in our framework.

Figure 4.5: Recorded Information in a Zipkin Span, Stored in JSON

Each trace contains several such spans with a common traceId. This particular capture corresponds to the instrumentation shown in Figure 4.3.


```
{
    "traceId": "4fcc591da1682e0f",
    "id": "4fcc591da1682e0f",
    "kind": "SERVER",
    "name": "post /todos",
    "timestamp": 1589750503288975,
    "duration": 24134,
    "localEndpoint": {
        "serviceName": "todos-api",
        "ipv4": "172.18.0.6"
    },
    "tags": {
        "http.path": "/todos",
        "http.status_code": "200",
        "req": "{\"url\":\"/todos\",\"method\":\"POST\",\"user\":{\"exp\":1590009700,\"firstname\":\"John\",\"lastname\":\"Doe\",\"role\":\"USER\",
            \"username\":\"johnd\"},\"body\":{\"content\":\"zsdasd\"}}",
        "req_headers": "{\"accept-language\":\"en-US,en;q=0.9,hi;q=0.8,it;q=0.7\",\"accept-encoding\":\"gzip, deflate, br\",\"referer\":\"http://localhost:8080/
            \",\"sec-fetch-mode\":\"cors\",\"sec-fetch-site\":\"same-origin\",\"origin\":\"http://localhost:8080\",\"user-agent\":\"Mozilla/5.0 (Macintosh;
            Intel Mac OS X 10_15_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.163 Safari/537.36\",\"sec-fetch-dest\":\"empty\",
            \"x-b3-sampled\":\"1\",\"x-requested-with\":\"XMLHttpRequest\",\"x-b3-spanid\":\"4fcc591da1682e0f\",\"accept\":\"application/json, text/plain, */*\",
            \"content-type\":\"application/json;charset=UTF-8\",\"authorization\":\"Bearer
            eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1OTAwMDk3MDAsImZpcnN0bmFtZSI6IkpvaG4iLCJsYXN0bmFtZSI6IkRvZSIsInJvbGUiOiJVU0VSIiwidXNlcm5hbWUiOiJqb2hu
            ZCJ9.t9c9Zzubm_DDYMNf4UvMmpncIeZKr4hHlkXzIwuHmfc\",\"x-b3-traceid\":\"4fcc591da1682e0f\",\"content-length\":\"20\",\"connection\":\"close\",
            \"host\":\"localhost:8080\"}",
        "res": "{\"content\":\"zsdasd\",\"id\":3}",
        "testTag": "v1"
    },
    "shared": true
```

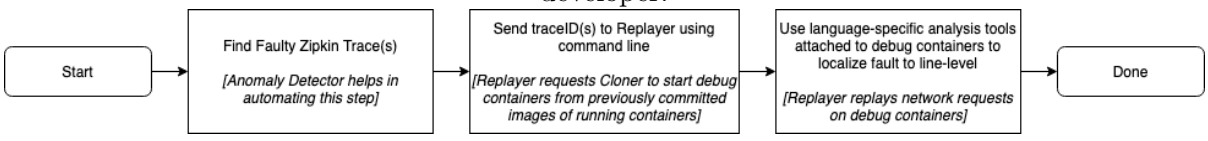

[5]https://cloud.google.com/container-registry

[6]https://hub.docker.com/

The Replayer service is implemented as a Python script that can replay traces in two modes: single trace replay mode or batch replay mode. In the single trace replay mode, the Replayer takes a trace ID as input. It first obtains the recorded information of that trace from the Zipkin API. This recorded trace information is in JSON format and has a set of span objects as shown in Figure 4.5. After parsing the trace data, the Replayer requests Cloner to start debug containers using images of production containers (that were committed manually or on a schedule) for each of the services in the trace. Using the `req` field of the tag information recorded in the first span of a trace, the root request is reconstructed using the Python requests library and emitted to the relevant debug container. Figure 4.7 shows the method of Replayer that replays a trace, given a trace id. This mode is useful for finding the root cause when some fault has already been localized to the trace level (which can be done by the Anomaly Detector). In the batch mode, a time-ordered set of Trace IDs is the input, the Replayer requests the Cloner to start debug containers of services, and then the Replayer emits requests associated with each trace at time intervals consistent with the original request. In both modes, the developer could see the original response and the replayed response. The developer can also see the original systems statistics (i.e. the memory, cpu, network usage) and the same statistics for the debug containers using cAdvisor. By replaying traces on debug containers, a developer could reproduce fault symptoms and potentially find root causes of faults at the line-level by following the steps shown in Figure 4.6. Case studies for finding specific root causes using our framework are mentioned in Section 5.2.

Figure 4.6: Steps for Replay Debugging

The text in non-italics indicates the steps a developer should take for replay debugging. Text in italics indicates the functionalities of the components of the framework for assisting the developer.



The Replayer can also read the `non_deterministic` flag on the traces and either ignore replaying those traces or perform a custom replay if the developers write handlers in the Replayer for a particular span.

Figure 4.7: Methods of REPLAYER for Replaying a Trace

```python
def start_debug_containers(services):
    """
    Calls Cloner to start debug container for each service
    """
    for service in services:
        os.system('python3 ../cloner/run_debug_container.py {}'.format(service))


def replay_trace(trace_id):
    """
    replays the trace on debug containers and outputs
    the recorded response and replayed response.
    """
    trace = get_trace_data_by_id(trace_id)  # get trace data from Zipkin API
    services = get_services_by_trace(trace)
    start_debug_containers(services)
    span = trace[0]
    if span.get("tags").get("req") is not None:
        req = {}
        try:
            req = json.loads(span["tags"]["req"])
        except:
            print("error in parsing recorded request")

        if bool(span["tags"].get('non_deterministic')) == True:
            print('skipping replay of non-deterministic request')
            return

        if req.get('url') is not None:
            headers = json.loads(span["tags"]["req_headers"])
            # base_url: address of the machine running debug containers
            url = base_url + req['url']
            req_type = req['method']
            params = {'body': req['body']}
            try:
                if req_type == 'GET':
                    res = requests.get(
                        url, params=params, headers=headers)

                elif req_type == 'POST':
                    res = requests.post(
                        url, data=req["body"], headers=headers)

                # print recorded and replayed response
                print_recorded_replayed(span['name'], json.loads(
                    span["tags"]['res']), json.loads(res.content))

            except:
                print('error in sending request')
    else:
        print("span did not have request information")

    return
```

4.4 Anomaly Detection

Anomaly detection is used in our framework for answering "What to replay to debug faults?". As mentioned earlier, finding the right trace to debug a fault is equivalent to the problem of finding a needle in the haystack since microservices typically produce a large number of traces which makes it hard to pinpoint problematic traces to find root causes of faults. Therefore, automating the search of traces associated with faults can enable the developer to debug much faster. The problem of anomaly detection in the context of finding faulty traces can be formally stated as follows:

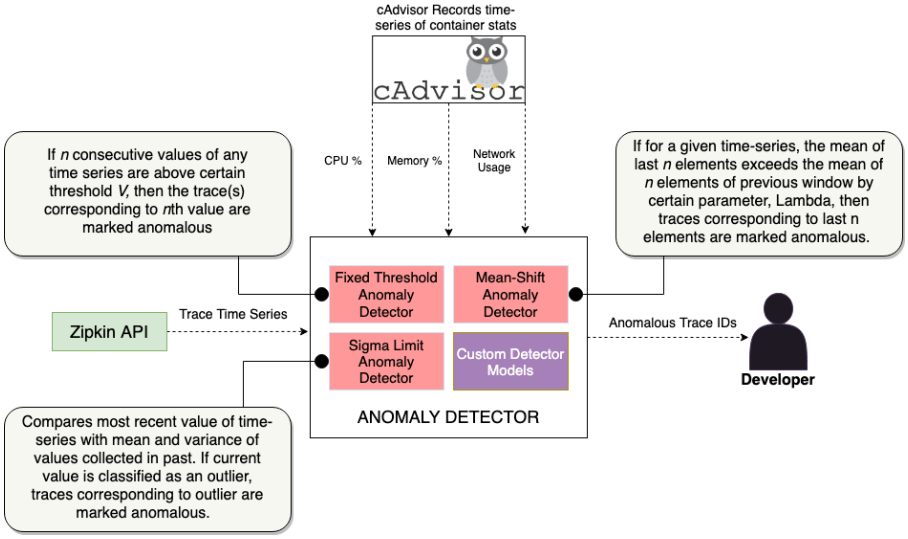
Given a time series of distributed traces, $\mathcal{T}$, obtained from running requests on a set of microservices $\mathbf{S}$, find a subset $F \subset \mathcal{T}$ such that replaying each trace $t \in F$ on replica services, $\mathbf{S}'$, can reproduce anomalies (i.e. fault symptoms) and therefore help in finding the root causes of faults. In an optimal solution, $|F|$ should be as small as possible while the replay of all $t \in F$ on $\mathbf{S}'$ should reproduce as many anomalies as possible, so that a small number of traces can be analyzed to find the root causes of maximum number of

latent faults.

We leverage the fault study of Chapter 3 for tackling the aforementioned problem of automatic identification of faulty traces, F, from the set of all traces, $\mathcal{T}$. Two microservice fault categories found in our fault study were: faults identified by delays/timeouts and faults identified by high resource usage. Since requests that caused an anomalous symptom (delay/timeout or high resource usage) also created a Zipkin trace, there is a temporal proximity of the trace and the anomaly. Therefore, automatic identification of delays/timeouts and high resource usages can help in identification of faulty traces, which in turn can lead to the root causes of the faults. With this insight, we design an Anomaly Detector service that automates the process of finding subset F by learning from historical traces and patterns of container resource utilization using statistical analysis techniques.

4.4.1 Anomaly Detector Implementation

Figure 4.8: Anomaly Detector Modules



The Anomaly Detector polls cAdvisor[7] during application run-time to obtain per-container resource usage time series (such as CPU%, Memory %, Network IO) that are used for anomaly detecion. Since the set of traces produced keeps increasing with time as a microservice application processes more client requests, the Anomaly Detector also polls the database of all traces using the Zipkin API for obtaining a fresh $\mathcal{T}$ after

[7]https://github.com/google/cadvisor

a certain configurable time interval. The time-series of per-span latencies obtained from $\mathcal{T}$ are also used for anomaly detection.

Essentially, the ANOMALY DETECTOR's responsibility is to observe the system for anomalous behaviour and then map that observation to a set of traces that can be used for finding the root cause. The mapping of an anomaly to traces is done using a configurable time window. For example, if the ANOMALY DETECTOR observes a sudden increase in memory usage % of a given container, it queries for the traces associated with that container in a time window of -1 second to +1 second of the increase and adds those traces to F. For prototyping this service, three statistical anomaly detection modules defined in [23] were used:

1. **Fixed Threshold Anomaly Detector Module**: this module monitors whether the values of a certain time series are crossing a fixed threshold and identifies traces that could have led to the crossing of the threshold. Formally, given a time series of values (eg. Memory usage % values) $v_1, ...v_p$ and a certain fixed threshold, V, if $v_{j+i} > V \ \forall i = 1...n$ for some $1 < j < p$ then all $t \in \mathcal{T}$ such that $(time(v_{j+n}) - \Delta) < time(t) < (time(v_{j+n}) + \Delta)$ are added to F. The configuration parameters for this detector are the fixed threshold, V (defined per service, per metric), the time window, Δ (around 1 second), and the number of consecutive values above threshold, n (defined per metric). This detector is good for detecting gradually increasing resource leaks or gradually increasing latencies.

2. **Sigma Limit Anomaly Detection Module**: this module detects sudden bursts or outliers in the values of a time-series and identifies traces that could have led to those bursts. Formally, given a time series of values $v_1, ...v_p$ and some sensitivity parameter σ, then v_j is anomalous if: $v_j - median(v_{j-i}, v_{j-i+1}...v_{j-1}) > \sigma \times std\_deviation(v_{j-i}, v_{j-i+1}...v_{j-1})$. All $t \in \mathcal{T}$ such that $(time(v_j) - \Delta) < time(t) < (time(v_j) + \Delta)$ are added to F. The configuration parameters for this detector are the sensitivity, σ(typically 3 - 5), mean window size, i, and time window, Δ (around 1 second). This detector is good for detecting sudden resource leaks or sudden latency increases due to specific inputs to services.

3. **Mean Shift Anomaly Detection Module**: this module detects long-term changes

in some quality indicator of services by comparing the most recent δ values of that indicator with the last δ values. Formally, given a time series of values $v_1, ...v_p$ and some sensitivity parameter λ, if $|mean(v_{p-\delta}, v_p) - mean(v_{p-2\delta-1}, v_{v-delta-1})| > \lambda$, then all $t \in \mathcal{T}$ such that $time(v_{p-\delta}) < time(t) < time(vp)$ are added to F. The configuration parameters for this detector are the sensitivity parameter, λ (defined per quality indicator), and window size, δ (a day or a week).

For prototyping the Anomaly Detector, the following time-series were used: span-level latency, container memory usage %, container CPU usage %, and network IO. For even better anomaly detection, the combinations of indicators could also be used. For example, the detection modules could be run on the time-series of ratio of memory consumption and the number of requests. Further, custom models defined by developers (such as [33]) could be integrated easily. The Anomaly Detector can also give a natural language reason for why particular traces were marked as faulty. For example, a developer could receive the following notification from the detector:

```
There was a sudden spike in memory usage by the todos-api service.
Traces: 00e26b1476bab62b and b74335f97997bcde were marked as faulty
by the Sigma Limit Anomaly Detector and may help in finding the root cause.
```

.

While the Anomaly Detector does not guarantee an accurate capture of the set of faulty traces, F, it is a helpful tool for a developer since it can often reduce the time it takes to localize a fault to the trace level.

CHAPTER 5

Evaluation

In this chapter we evaluate the framework described in Chapter 4 by answering the following research questions:

- ***Q1:*** *Can the proposed record and replay framework successfully reproduce common faults and symptoms and assist a developer in finding the root cause of the faults?*
- ***Q2:*** *What is the overhead of recording services as described in Section 4.2 and how does it compare to overheads of more faithful, language specific techniques of recording?*

We answer these questions by running experiments on an open-source microservices application described in the next section.

5.1 Application Used for Experiments

microservice-app-example[1] is a polyglot application that lets users login through a web browser and then create or delete TODO items. This application was chosen for running experiments because of a few reasons: 1) the code repository is freely available on Github and has 1.3k stars (a measure of popularity in the open-source community), 2) it is a complete CRUD web-application, 3) services of the application are written in four commonly used languages, 4) the application runs using popular multi-container build and run tool Docker, and Compose[2] 5) services have basic Zipkin instrumentation by default.

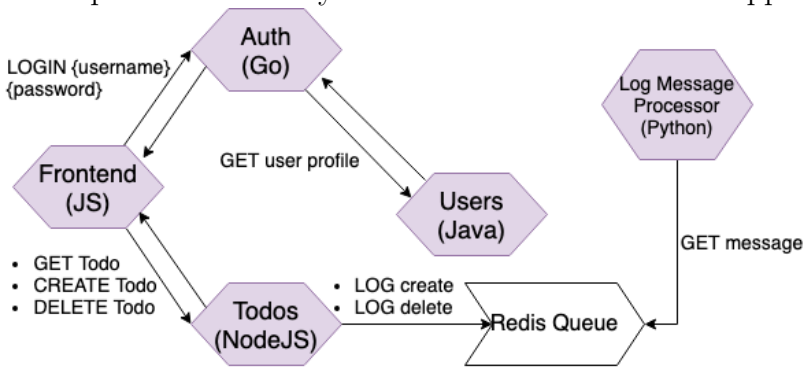
[1] https://github.com/elgris/microservice-app-example

[2] https://docs.docker.com/compose/

This application consists of 5 different services whose interaction is shown in Figure 5.1. At a high level, the services and their corresponding functionalities are:

- **auth-api**: Service written in Go for authorization. JSON Web tokens that are used with other APIs are generated.
- **todos-api**: Service written in NodeJS for managing a user's TODO records. This service sends 'create' or 'delete' log messages to a Redis queue.
- **users-api**: Service written in Java (Spring Boot application) for maintaining user profiles.
- **log-message-processor**: Service written in Python for listening to Redis queue and printing logs.
- **frontend**: UI service written in Javascript (VueJS) for letting users login, create and delete TODO items on a browser.

Figure 5.1: Requests initiated by each service in microservice-app-example



Before running experiments on *microservice-app-example*, the Zipkin instrumentation of services were modified to record the request and response as described in Section 4.2. Additionally, the cAdvisor image was added as a service in the application's configuration YAML so that resource usage and performance metrics of each service container could be recorded in real-time.

5.2 Qualitative Evaluation

For answering ***Q1***, some faults from Table 3.5 were inserted one at a time and the framework was used for replaying the fault in debug containers for finding the root cause. The following three case studies describe the fault and exact debugging steps required to find the root cause at the line level or function level.

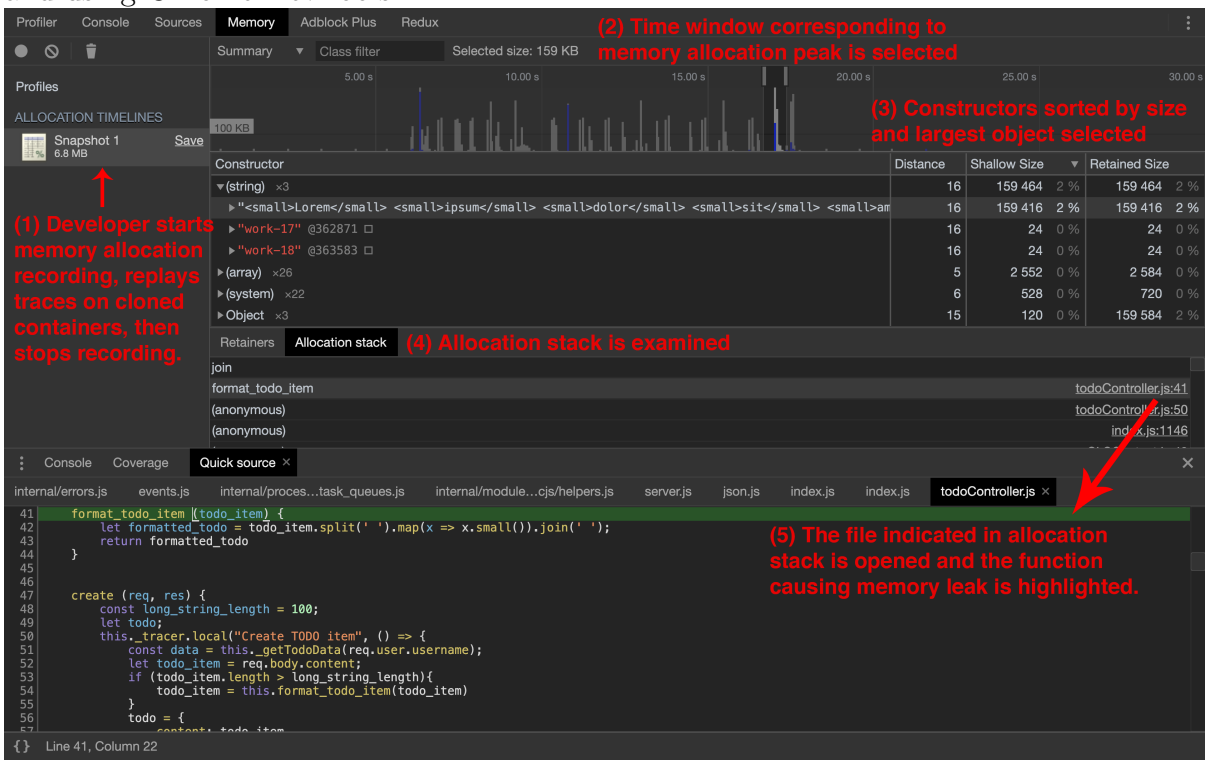
5.2.1 Case Study 1: Debugging a Latent Memory Leak

Fault Description and Scenario: A resource leak fault that can cause large memory allocations (mutation 9 from Table 3.5) was inserted in the *todos-api* service. This fault occurs when a user of the *frontend* service inputs a long string when creating a TODO item. Memory leaks only when a specific execution path is taken in the NodeJS service. This is a latent fault because the *todos-api* service does not crash unless several large TODO items are created, at which point the memory allocated to the container is insufficient. Therefore, debugging this fault before the service crashes is hard because a developer needs two things: 1) some signal that memory leaks are happening in a particular service, 2) the set of inputs causing the memory leak. For simulating a production-like workload for a TODOs application with this fault, several successive requests with varying TODO item lengths were made on the *frontend* service.

Replay Debugging Steps: The Anomaly Detector detects a sudden increase in memory consumption (using the sigma-limit anomaly module) by the *todos-api* container when a long TODO is created and notifies the developer. For debugging, the developer wants to replay requests on *todos-api* service. However, this cannot be done in production since replaying requests would result in non-user-initiated actions registered with production services. Therefore, the developer would want to replay requests in a debug environment. For doing this, the developer can request the Replayer to replay all traces that happened in a 20 second window of the anomaly notification. The Replayer gets the requests of the relevant recorded requests from the Zipkin API and requests the Cloner to create debug containers of services that are in any invocation chain with *todos-api*. The Cloner, which has cached version of service images at different timestamps, creates the relevant containers in a debug environment. The debug

todos-api service is run in the inspect mode and an SSH tunnel is automatically setup so that the developer can attach Chrome Node DevTools to the service running in the container.

Figure 5.2: Steps to Find Root Causes of NodeJS Memory Faults by Replaying Traces and using Chrome DevTools



Using Chrome Node DevTools' memory recording feature and the REPLAYER for replaying requests on the debug containers, the developer follows the 5 steps shown in Figure 5.2 to find the exact root-cause of the fault. It is worth mentioning that memory recording could not be directly done on production containers because that would result in a 3-4X latency overhead, making the *todos-api* service very slow.

The debugging technique used in this case study could also be used for finding the root-cause of mutation 10, if the debug containers are run by using the same configuration files as used for production containers.

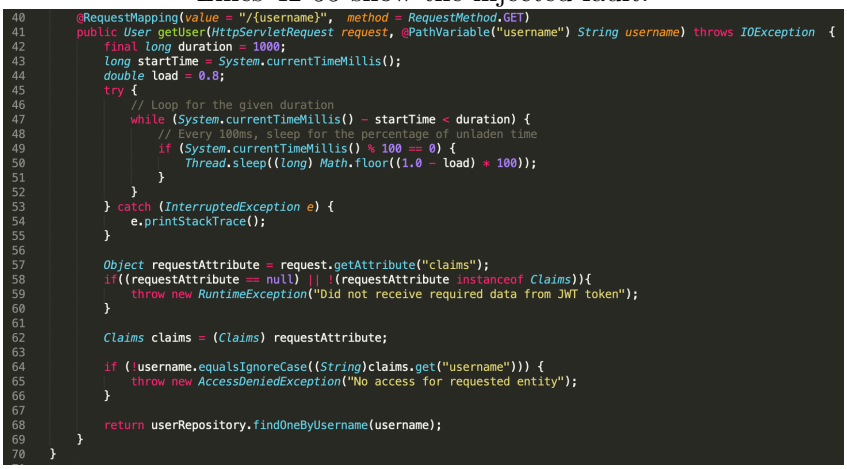
5.2.2 Case Study 2: Debugging a Latent High CPU usage

Fault Description and Scenario: A fault that causes high CPU usage (mutation 11 from Table 3.5) was inserted in the *users-api* service by modifying the Java source

code and redeploying the service. This fault is triggered when a user enters their login credentials on the *frontend* service running on a browser. The *frontend* service requests the *auth-api* service for authentication, which in turn sends a request to *users-api* which runs a CPU intensive while loop in the `getUser()` method of `UsersController` class. The fault code is shown in Figure 5.3. This is a latent fault, since all services respond without throwing any errors, while the end user experiences a slight delay in logging in.

Figure 5.3: Mutation in *users-api* Causing High CPU Usage

Lines 42-55 show the injected fault.

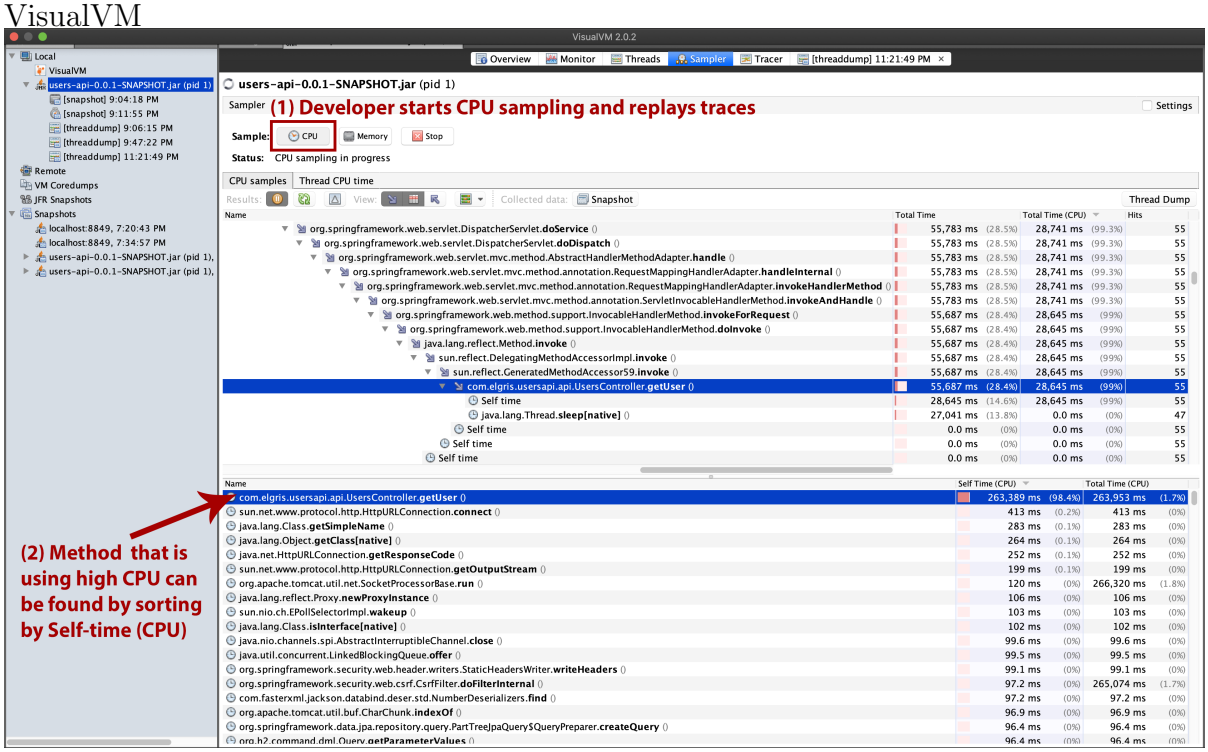


```
40      @RequestMapping(value = "/{username}",  method = RequestMethod.GET)
41      public User getUser(HttpServletRequest request, @PathVariable("username") String username) throws IOException  {
42          final long duration = 1000;
43          long startTime = System.currentTimeMillis();
44          double load = 0.8;
45          try {
46              // Loop for the given duration
47              while (System.currentTimeMillis() - startTime < duration) {
48                  // Every 100ms, sleep for the percentage of unladen time
49                  if (System.currentTimeMillis() % 100 == 0) {
50                      Thread.sleep((long) Math.floor((1.0 - load) * 100));
51                  }
52              }
53          } catch (InterruptedException e) {
54              e.printStackTrace();
55          }
56
57          Object requestAttribute = request.getAttribute("claims");
58          if((requestAttribute == null) || !(requestAttribute instanceof Claims)){
59              throw new RuntimeException("Did not receive required data from JWT token");
60          }
61
62          Claims claims = (Claims) requestAttribute;
63
64          if (!username.equalsIgnoreCase((String)claims.get("username"))) {
65              throw new AccessDeniedException("No access for requested entity");
66          }
67
68          return userRepository.findOneByUsername(username);
69      }
70  }
```

Replay Debugging Steps: Once a developer is notified that user logins have been slower (notification could be given by Anomaly Detector), they would want to find the root cause without disrupting any service. The root-cause could be found by following the replay debugging steps shown in the flowchart in Figure 4.6. First, the developer would obtain recent traces having the login flow by querying Zipkin. The trace ids of the obtained traces are forwarded to the Replayer. The Replayer requests the Cloner to start debug containers attached with language-specific debugging tools. The debug *users-api* service is run with debug flags and an SSH tunnel is automatically setup so that the developer can attach VisualVM to the service running in the container. As shown in Figure 5.4, by replaying the network requests on debug containers and by using the CPU sampling feature of VisualVM, the method taking the most CPU time can be found. It is important to note that VisualVM could not be used for CPU profiling a production container since that would significantly slow down an already slow service, and disrupt end user experience.

The debugging technique used in this case study could also be used for finding the

Figure 5.4: Steps to Find Root Causes of Java Faults by replaying traces and Using VisualVM



root-cause of mutation 12 if the debug containers are run by using the same configuration files as used for production containers.

5.2.3 Case Study 3: Debugging Slow Responses

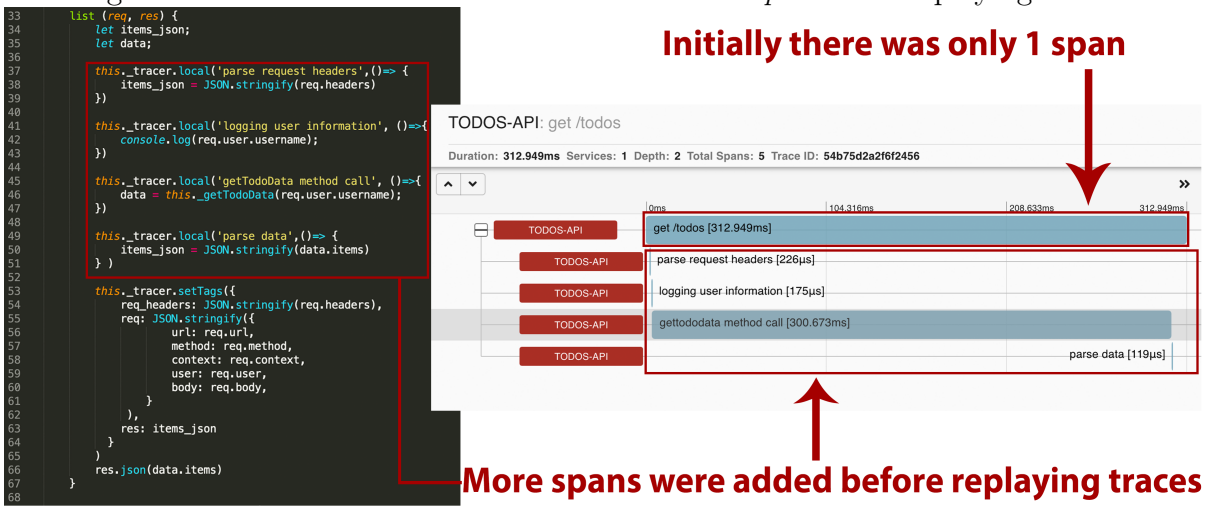
Fault Description and Scenario: A fault that adds a consistent delay in response (mutation 1 from Table 3.5) was inserted in the *todos-api*. The JavaScript line "if (userID == 'johnd') sleep(300)" (sleep(n) delays execution by n milliseconds) was inserted in a method that obtains a user's data from the cache. This fault line makes the `/todos` endpoint slower for only one user by 300 milliseconds. For simulating a production-like workload for a TODOs application with this fault, several login, create post, view post, and delete post requests were made on the *frontend* service using different userids.

Replay Debugging Steps: In this case study, the sigma-limit anomaly detection module for per-span latency was configured with $\sigma = 4$ and $i = 25$. This means, that any trace whose any span has a latency higher than 4 times the median latency of previous 25 same spans would be marked anomalous. Using the sigma-limit anomaly module, ANOMALY DETECTOR detects a sudden increase in latency of the `/get todos` span

when user `johnd` accesses the `/todos` endpoint and sends the trace associated with this anomaly to the developer. The developer looks at the top level function associated with the `/get todos` span, but is unable to figure out why the latency is high for the captured trace. The developer uses the CLONER to create debug containers, SSHs into the *todos-api* debug container and adds more spans within the top level function by modifying the source as shown in Figure 5.5. The debug containers are run, the REPLAYER is used to replay the captured traces and the developer sees that `getTodoData` method call is taking the most time using the Zipkin web UI. The developer could now examine this function and find the root-cause.

It is important to note that adding finer grained spans in production is not a good idea because latency increases linearly with the number of spans (Figure 5.7). This case study shows that finer-grained instrumentation can be done within debug containers before replaying traces. Using finer grained instrumentation during replay, we could also find the root-causes of mutations 3, 4, 13, and 15.

Figure 5.5: More Instrumentation Done in *todos-api* before Replaying Traces



5.3 Performance Evaluation

We evaluate the performance in terms of latency overheads added by instrumentation, since latency is the most important metric for a service's performance and is often defined in SLAs. We also measure the time during which a container is paused for cloning.

Latency Overheads of Zipkin					
Service	**Route or Function called**	**Statistic**	**Latency (ms) - No Tracing**	**Latency (ms) - Zipkin tracing**	**Overhead %**
todos-api (Javascript)	/create (creates a todo-item)	Median	7.7950	8.1530	4.59%
		95th Percentile	13.1103	12.9318	-1.36%
	/delete (deletes a todo item)	Median	7.1690	8.6160	20.18%
		95th Percentile	11.8577	15.8530	33.69%
auth-api (Go)	/login (authorization by generating JWT tokens)	Median	15.8365	16.6755	5.30%
		95th Percentile	22.8493	24.1086	5.51%
user-api (Java)	/user (provides user profile)	Median:	3.3778	5.7045	68.88%
		95th Percentile:	8.0304	11.7328	46.11%
log-message-processor (Python)	log_message (listens to redis queue and prints to console)	Median	0.1287	6.9149	5270.93%
		95th Percentile	0.4418	12.3511	2695.92%

Table 5.1: Latency Overheads of Recording Zipkin Traces for Sample Microservices Application

5.3.1 Latency Overheads

For answering ***Q2***, first, all instrumentation was removed from each service so that no Zipkin traces were created on requests. 1000 successive requests, with very small random delays (< 200ms) in between, were made on each service and latency of getting complete response was recorded. Then, the instrumentation prescribed by the framework was added and the same 1000 requests were sent to each service again. Table 5.1 shows the median and 95th percentile latencies with and without instrumentation and the overhead percentage.

The minimum overhead for the median request latency with instrumentation % was 4.59% for the /create route of the *todos-api* service. The outlier in terms of overhead was *log-message-processor*. The reason for excessive overheads in this service is that without any instrumentation, the log_message function of this service simply listens to a redis queue and prints to console, whereas, with instrumentation this function has to set up an HTTP connection which requires significantly more time than printing to console.

For measuring the overhead of more faithful recording techniques, the same 1000 requests were sent on a new container running *todos-api* but this time with Chrome NodeJS Devtools memory recording on. Table 5.2 contains the measured latency and overhead % and it can be seen that with memory recording, there was 3X overhead for

the median request and 4.6X overhead for the 95th percentile request. This is an order of magnitude more latency overhead than the Zipkin instrumentation overhead for the same service and routes.

Route Called	Statistic	Latency - No Tracing (ms)	Latency - Memory Recording (Chrome DevTools)	Overhead %
/create (creates a todo-item)	Median	7.7950	31.7100	306.80%
	95th Percentile	13.1103	74.1608	465.67%
/delete(deletes a todo item)	Median	7.1690	28.1690	292.93%
	95th Percentile	11.8577	67.0037	465.06%

Table 5.2: Latency Overheads of Using NodeJS DevTools for Recording Memory Allocation of 'todos-api'

Latency overhead due to Zipkin instrumentation depends on several factors including programming language/framework choice, function execution time, how many spans are used in recording, and the physical distances between requesting, responding and tracing services. To illustrate the latency overhead's dependence on programming language/framework choice, two simple servers were created in Python (using Flask) and NodeJS (using Express) and instrumented using the respective Zipkin instrumentation libraries. Both the servers simply returned a simple string response after sleeping for a certain number of milliseconds. Response time for non-instrumented servers had negligble difference for the servers, however after adding instrumentation, there was significant difference in overheads. Figure 5.6 shows the relationship of overhead % of 95th percentile latency and function execution time for the Python and NodeJS servers. To illustrate the relationship of latency with number of spans in a trace, an endpoint in the Flask server that takes 100ms for execution was created. Different number of spans were used and 100 requests were made to that endpoint for each number of spans. Figure 5.7 shows that each addition of a span adds about 1-2 milliseconds of latency. In the figure, the latency corresponding to 0 spans is the time taken when there is no tracing at all.

Figure 5.6: P95 Latency Overhead of Zipkin Instrumentation % versus Function Execution Time

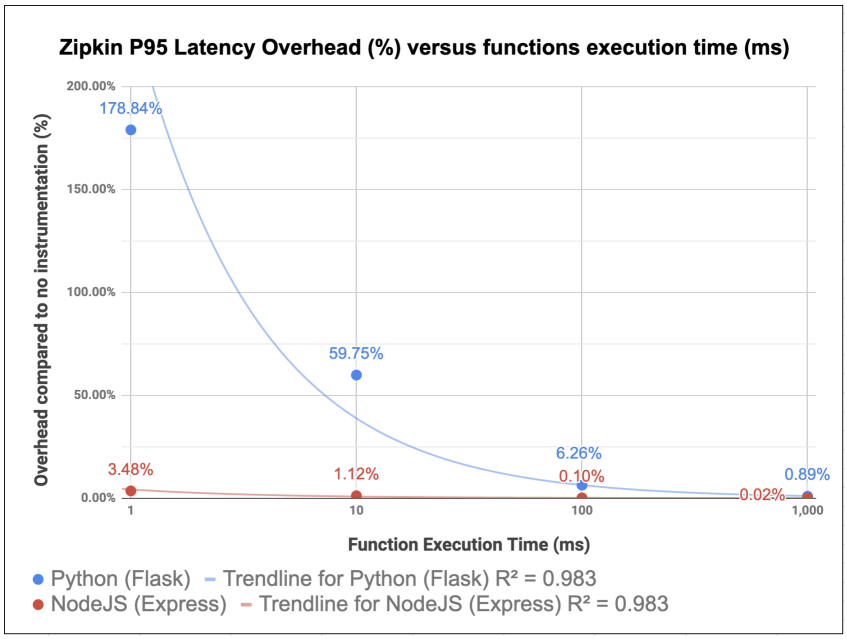
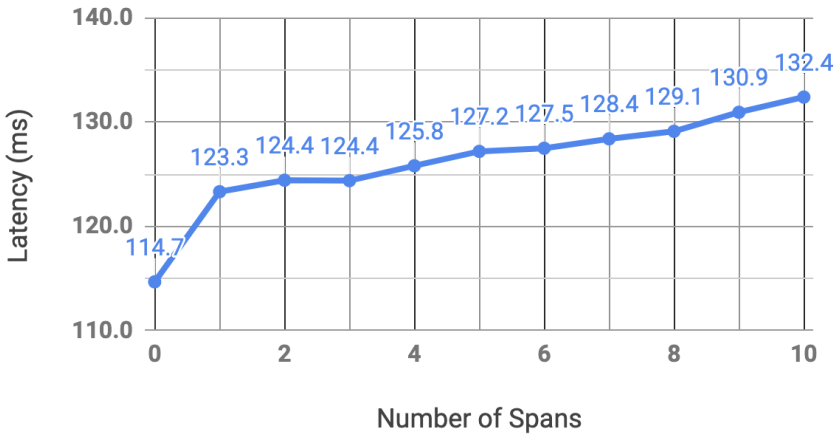


Figure 5.7: Latency versus Number of Spans in Endpoint



Container Commit Time		
Service	**Image Size**	**Commit Time**
todos-api (Javascript)	88.3MB	5.2790ms
auth-api (Go)	354MB	5.4350ms
user-api (Java)	267MB	4.9830ms
log-message-processor (Python)	285MB	4.6380ms
frontend (Javascript)	245MB	4.1530ms

Table 5.3: Container Commit Time for Different Services of Benchmark Application

5.3.2 Container Cloning Overhead

During container cloning using Docker Commit[3], the container being committed is paused for preventing data corruption during the commit. Therefore, it is necessary that the clone process is very fast so that services are not disrupted. For measuring the time to commit a container, all the containers of microservice-app-example were run, a sample work load was run and then each container was committed by the Cloner. As seen in Table 5.3, the commit times for each service are between 4-6ms. Since commit time is less than response times for requests on each service, containers could be committed without disrupting services because requests would not drop if the containers are paused for a few milliseconds.

5.4 Summary

In this chapter we evaluated our record and replay debugging framework using an open-source microservices application. The three case studies demonstrate how this framework can be used to debug latent faults such as memory leaks, high CPU usage, or slow responses by replaying network requests on cloned containers with heavy debugging tools attached. Using the techniques described, nine out of fifteen mutations can be debugged. The other six mutations (# 2, 5, 6, 7, 8, 14) may also be reproduced in the debug environment using this framework, though it would be more complex to localize the fault. For instance, configuration mutations are hard to debug precisely to line level

[3]https://docs.docker.com/engine/reference/commandline/commit/

without expert knowledge of how everything is configured. Even the usage of heavier analysis or debugging tools may not lead to the exact line in the configuration files that should be fixed.

We also evaluated the performance of our framework in terms of the time during which a container is paused for cloning and latency overheads added by Zipkin instrumentation. The pause times for committing containers (of microservices-app-example) using Docker commit was 4-6ms. Latency overheads of instrumentation specified by the framework, in the best case were about 1-4ms. In the worst case (in which there are no other network calls other than the tracing call), the latency overheads can be >10ms. The high overheads could potentially be reduced by modifying the Zipkin implementation to send network requests asynchronously. We also showed that latency overheads depend on programming language/framework choice, and the number of spans within a trace. Our framework prescribes creating 1 span per endpoint, and therefore adds minimal latency due to number of spans. It is important to note that most of the overheads due to distributed tracing already exist in industrial microservice applications that use systems like Zipkin or Jaegar. Further, the aforementioned evaluations show that the overheads of our framework can be 10X less than the case in which language specific recording tools (such as NodeJS Dev Tools or VirtualVM) are used directly on production services to record executions.

CHAPTER 6

Conclusion

As industrial software systems shift increasingly towards microservice architectures, there is a need for novel debugging tools and techniques that tackle unique challenges of this architecture such as the polyglot nature of the system, large accumulated states, and tight latency limits. In this work, we studied 25 faults from real-world microservices applications and observed that a majority of faults can be reproduced by replaying network requests on containers in the same state as the production container's state at the time of recording. We then designed a language-agnostic record and replay debugging framework for microservice applications that uses a distributed tracing system such as Zipkin for recording network requests and enables replay of those requests on cloned debug containers with heavier analysis tools such as Chrome DevTools or VisualVM attached. While our framework cannot reproduce all executions of a microservices application, it can be a useful tool for finding root-causes of common faults. Further, our framework specifies how to proactively find a needle (faulty trace) in a haystack (set of all traces produced) using span-level and container-level monitoring to detect fault symptoms found in our study. In our experiments on an open-source microservices application, we found that our framework can indeed simplify the debugging process and therefore is a practical approach for enabling replay debugging in real-world microservice applications.

APPENDIX A

Descriptions of Microservice Application Faults

This appendix provides a description of each of the faults collected in our fault study. The description includes some background context about the fault and the root-cause of the fault. The level of detail recorded here is different for different faults because the sources of faults are diverse (i.e. published research, Stack Overflow, Github issues) due to which the information available per fault varied. The following subsections of fault descriptions are organized by fault symptoms (as defined in Chapter 3).

A.1 Fault Symptom: Errors and Exceptions

- **F1. Fault Context:** This fault was observed in the Clearwater IP Multimedia Subsystem (IMS) microservices system. The deletion of telephone accounts was attempted and two 502 responses were triggered.

  Debugging Steps and Root Cause: Debugging started when developers were troubleshooting a failure reported by a Clearwater client. The developers examined logs of several candidate microservices. One service had anomalies relating to the telephone accounts, and further examination showed error in interaction with Cassandra database.

  Source: Microservices Monitoring with Event Logs and Black Box Execution Tracing (Pg 4) [5]

- **F2. Fault Context:** SiteWhere provides a microservices platform for Internet of Things applications. In this fault, one of the services of SiteWhere was sending state updates even after being terminated due to which exceptions were thrown.

  Debugging Steps and Root Cause: Exceptions being thrown by a terminated service were observed. The root-cause was that the updates thread was not be-

ing terminated on shutdown of the service. The fix was to correctly handle the shutdown of the state updates thread.

Source: https://github.com/sitewhere/sitewhere/issues/726

- **F3. Fault Context:** Fault in SiteWhere: an unhandled exception is thrown when an HTTP POST request is made to a particular route.

  Debugging Steps and Root Cause: Debugging started when a user encountered this fault and created an issue on Github. The root-cause was that the API contract of that route was not clearly defined. Because of this, the user's POST request had a missing field in the body.

  Source: https://github.com/sitewhere/sitewhere/issues/607

- **F4. Fault Context:** Fault found by surveying a Senior Software Engineer of company that creates a travel assistance system. The fault symptom was that the payment service of the application fails.

  Debugging Steps and Root Cause: Six microservices were involved in the fault. Visual trace analysis was used to find that the root cause was that there was an overload of requests to a third-party service, which lead to a DoS (Denial of Service). Total time spent in debugging was 16 hours.

  Source: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study, TSE'18 [31].

- **F5. Fault Context:** An developer of a Spring Boot microservices application was getting an error (ConfigServicePropertySourceLocator : Could not locate PropertySource: I/O error on GET request) when executing the client app. The developer suspected that the settings they specified in their configuration file were not being used during execution.

  Debugging Steps and Root Cause: A question about the fault was asked on StackOverflow by the developer. Another user responded by saying that the configuration property `spring:cloud:config:enabled: true` should be added to the files application.yml and bootstrap.yml, which fixed the fault. Thus, the root-cause was that a parameter was not configured in the .yml files.

Source: https://stackoverflow.com/questions/37074642/settings-in-application-yml-for-spring-cloud-config-arent-used-when-app-is-exec

- **F6. Fault Context:** A developer working on a Spring Boot microservices application gets an "Unauthorized" error when they access information about traces and logs of services.

  Debugging Steps and Root Cause: A question was asked on StackOverflow about the fault and a few different solutions were provided by users. The solution that got the most upvotes was configuring the management.security.enabled=false flag in the application.properties.

  Source: https://stackoverflow.com/questions/42648060/unauthorized-in-spring-boot-admin

- **F7. Fault Context:** A developer working on a microservices application with Zuul, a dynamic routing and monitoring gateway service gets an internal server error (HTTP Code 500) when services are run on Docker.

  Debugging Steps and Root Cause: A question was asked on StackOverflow about the fault. A user diagnosed the problem and gave a solution: each service deployment should be explicitly connected to the same Docker network and Zuul's properties.yml should have paths configured.

  Source: https://stackoverflow.com/questions/55026430/zuul-forwarding-error-internal-server-error-500

- **F8. Fault Context:** A developer running a Spring and OAuth2 sample microservices application available on a tutorial site was getting TransportExceptions.

  Debugging Steps and Root Cause: A question was asked on StackOverflow about the fault and it has been viewed over 50,000 times since then. The root cause was that the application was attempting to automatically register with the Eureka service, which was unintended behaviour. To fix this, two lines (eureka.client.register-with-eureka=false and eureka.client.fetch-registry=false) had to be added to the application.properties configuration.

Source: https://stackoverflow.com/questions/55026430/zuul-forwarding-error-internal-server-error-500

- **F9. Fault Context:** An error is encountered by a user of New York Times' open-source microservices toolkit, Gizmo. When the 'prefix' value of a service is empty, there is a silent failure.

  Debugging Steps and Root Cause: An issue was created on Github. The root-cause was that the 'prefix' attribute of a service which should be optional was instead required. So the users who set 'prefix' to an empty value, were running into silent failures. The fix was to do a refactoring that would remove the Prefix() method from the server.Service interface and add it to the config.Server.

  Source: https://github.com/nytimes/gizmo/issues/26

A.2 Fault Symptom: Delays or Timeouts

- **F10. Fault Context:** A timeout of the server (504 error code) was experienced by a user of the Clearwater IP Multimedia Subsystem (IMS) microservices system when they were attempting a voice telephone call.

  Debugging Steps and Root-Cause: Log inspection showed the faulty microservice has been unavailable in close time proximity to the timeout experienced by the client. The developers needed apriori knowdlege of service architecture to figure out that the root cause was that the connection was refused by Cassandra database (call in 3rd degree service), which is used to store authentication credentials and profile information in Clearwater.

  Source: Microservices Monitoring with Event Logs and Black Box Execution Tracing (Pg 4) [5]

- **F11. Fault Context:** MongoDB server selection timeout exceeded in SiteWhere, a microservices platform for Internet of Things applications. **Debugging Steps and Root-Cause:** A user of SiteWhere observed this fault when their microservices started before the MongoDB replica initialized and created an issue on Github. The root-cause was that the server selection timeout was configured to 30 seconds.

Changing the configuration to have an indefinite wait time fixed the fault.

Source: https://github.com/sitewhere/sitewhere/issues/721

- **F12. Fault Context:** This is a production bug in a running MySQL instance. Extremely long delays were observed when the mysql client was requested to execute long INSERT statements with different character sets. This bug was recreated in [1] with a client-server setup for evaluating Parikshan.

  Debugging Steps and Root-Cause: The MySQL server was cloned by Parikshan and then complex scripts (in Chinese and Japanese) were sent to the production MySQL server, which were asynchronously replicated in the debug container. SystemTap was used on the debug containers for pinpointing functions causing the slow-down. The root-cause was that there were multiple calls in a loop which caused the slow-down.

  Source: https://bugs.mysql.com/bug.php?id=15811 (Reported in [1])

- **F13. Fault Context** While using redis-2.6.11, a slave is unable to synchronise with a master.

  Debugging Steps and Root Cause: Failure observed when there was an attempt to synchronize a big database but the slave timed out. *Parikshan* was used to create debug containers and debuggers were used during replay. The root-cause was that a lower output buffer limit should have been specified in the configuration with: client-output-buffer-limit slave 4096mb 2048mb 60

  Source: https://github.com/antirez/redis/issues/957 (Reported in [1])

- **F14. Fault Context** Multiple faults in benchmark microservice application, *Social Network*. Quality of Service (latency/throughput) violations in several services.

  Debugging Steps and Root Cause: Seer, a system that uses models trained on trace data for detecting QoS violations was used for automated debugging. The root-causes were: resource contention causing violations in the memached service and long synchronization times causing violations in Thrift services.

  Source: Seer, ASPLOS'19 [12]

- **F15. Fault Context:** This fault was injected to a Java microservices application, *Guestbook*, for showing that the author's root cause analysis system works. The fault injection logic activates once per hour, and slows down all datastore invocations by 45ms for a period of 3 minutes

  Debugging Steps and Root Cause: The Service Level Objective (SLO) based anomaly detector of monitoring system, Roots, was run with sampling rate of 15 seconds. Anomalies were detected correctly and the root-cause was that the AppScale datastore service was slow to respond.

  Source: Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications, WWW'17 [16].

- **F16. Fault Context:** Fault found by surveying an architect of company that creates a mobile payment system. The fault symptom was that a particular service slows down and eventually returns an error.

  Debugging Steps and Root Cause: Eight microservices were involved in the fault. Visual trace analysis was used to find that the root cause was that there were recursive requests to a microservice due to SQL errors in another dependent service. Total time spent in debugging was 24 hours.

  Source: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study, TSE'18 [31].

A.3 Fault Symptom: Unusual Resource Usage

- **F17. Fault Context:** This fault was described in a sample scenario of a multi-tier SOA application that has an Nginx server and a Glassfish server, among other services running in separate containers. The symptom observed is unusual memory usage by the Glassfish service due to which error logs are generated in the Nginx server.

  Debugging Steps and Root Cause: Debugging starts when developers notice the unusual memory usage and surmise that the underlying fault is a memory leak. The developers use *Parikshan* to create Nginx-debug and Glassfish-debug contain-

ers and to forward network requests from the production containers to these debug containers. The developers then use heavier instrumentation on debug containers (using their preferred tools) to find that the root-cause is a persistent memory leak in the Glassfish application.

Source: Replay without Recording of Production Bugs for Service Oriented Applications [1]. Page 2.

- **F18. Fault Context:** Error in New York Times' open-source microservices toolkit, Gizmo. A memory leak is observed in a specific version (1.7)

  Debugging Steps and Root Cause: An issue was created on Github. The root-cause was that Gizmo's router implementation uses a registry (gorilla/context) for global request variables that never gets cleared in version 1.7, and thus there is a memory leak.

  Source: https://github.com/nytimes/gizmo/issues/74

- **F19. Fault Context:** A redis-2.4.9 master and slave are setup as separate services. Running concurrent requests through the client triggers memory leaks.

  Debugging Steps and Root Cause: *Parikshan* is used to create debug containers for the master and slave. Debug tracing is turned on, concurrent requests are replayed and the root cause is found to be a memory leak in master.

  Source: https://github.com/antirez/redis/pull/417 (Reported in [1])

- **F20. Fault Context:** Fault in SiteWhere, a microservices platform for Internet of Things applications. Loading an invocation in InfluxDB by unique id was overloading RAM.

  Debugging Steps and Root Cause: An issue was created on Github by a contributor of SiteWhere after they received a bug report from a user. The root-cause was that the InfluxDB command invocation query was inefficient because all event data was being loaded in the RAM, instead of only the data needed.

  Source: https://github.com/sitewhere/sitewhere/issues/655

A.4 Fault Symptom: Unexpected Output

- **F21. Fault Context:** Fault found by surveying a Staff Software Engineer of company that creates an online meeting system. The fault symptom was that certain messages were being displayed in wrong order.

  Debugging Steps and Root Cause: Three microservices were involved in this fault. Basic log analysis was used to figure out that the root-cause was asynchronous message delivery lacking sequence control. Total time spent in debugging this fault was 56 hours.

  Source: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study, TSE'18 [31].

- **F22. Fault Context:** Fault found by surveying a Senior Software Engineer of company that creates a collaborative translation system. The fault symptom was that some information was displayed incorrectly in a report.

  Debugging Steps and Root Cause: Six microservices were involved in this fault. Visual log analysis was used to identify that the root cause was "different data requests for the same report are returned in an unexpected order". Total time spent in debugging was 26 hours.

  Source: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study, TSE'18 [31].

- **F23. Fault Context:** Fault found by surveying a manager of a company that creates an Office Automation system. The fault symptom was that there was an unexpected change in the default selection on a web page.

  Debugging Steps and Root Cause: Three microservices were involved in this fault. Basic log analysis was used to find the root-cause that the key in the request of a microservice was not being passed to another dependent microservice. Total time spent in debugging was 40 hours.

  Source: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study, TSE'18 [31].

- **F24. Fault Context:** Fault found by surveying a Senior Software Engineer of a company that creates an Product Data Management System. The fault symptom was that "the bill of material (BOM) tree of a product is erroneous after updates".

  Debugging Steps and Root Cause: Six microservices were involved in this fault. Visual log analysis was used to find that the root-cause was updating the BOM data in an unexpected sequence. Total time spent in debugging was 32 hours.

  Source: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study, TSE'18 [31].

- **F25. Fault Context:** A bug was encountered in an open-source microservices project, open-loyalty, a platform for gamification and other loyalty features. The fault occurs on the admin page of the application when a user tries to edit a customer's level, some levels are missing in the dropdown menu.

  Debugging Steps and Root Cause: The root cause of the bug was that a service endpoint was being called with a missing argument from the admin service. By changing the route requested from api/level to api/level?perPage=total_level, this fault could be fixed.

  Source: https://github.com/DivanteLtd/open-loyalty/issues/78

REFERENCES

[1] Nipun Arora et al. "Replay without recording of production bugs for service oriented applications". In: *ASE 2018*. 2018.

[2] Kristin Brennan. "Microservices Trends Report". In: (2018). URL: https://lightstep.com/blog/microservices-trends-report-2018/.

[3] Brian Burg et al. "Interactive Record/Replay for Web Application Debugging". In: *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*. UIST '13. St. Andrews, Scotland, United Kingdom: Association for Computing Machinery, 2013, pp. 473–484. ISBN: 9781450322683. DOI: 10.1145/2501988.2502050. URL: https://doi.org/10.1145/2501988.2502050.

[4] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey". In: *ACM computing surveys (CSUR)* 41.3 (2009), pp. 1–58.

[5] M. Cinque, R. Della Corte, and A. Pecchia. "Microservices Monitoring with Event Logs and Black Box Execution Tracing". In: *IEEE Transactions on Services Computing* (2019).

[6] Docker. "Why Docker?" In: (2020). URL: https://www.docker.com/why-docker.

[7] Nicola Dragoni et al. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

[8] IBM Cloud Education. "Containerization". In: (2019). URL: https://www.ibm.com/cloud/learn/containerization.

[9] André Fachat. "Challenges and Benefits of the Microservices Architectural Style". In: (2019). URL: https://developer.ibm.com/articles/challenges-and-benefits-of-the-microservice-architectural-style-part-1/.

[10] Cloud Native Computing Foundation. "Jaeger". In: (2020). URL: https://www.jaegertracing.io/.

[11] Martin Fowler. “Microservices”. In: 2014. URL: https://martinfowler.com/articles/microservices.html.

[12] Yu Gan et al. “Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: ACM, 2019, pp. 19–33. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304004. URL: http://doi.acm.org/10.1145/3297858.3304004.

[13] Dennis Michael Geels et al. “Replay debugging for distributed applications”. PhD thesis. University of California, Berkeley, 2006.

[14] V. Heorhiadi et al. “Gremlin: Systematic Resilience Testing of Microservices”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. June 2016, pp. 57–66. DOI: 10.1109/ICDCS.2016.11.

[15] Jacob Jackson. “Debugging Microservices: Lessons from Google, Facebook, Lyft”. In: (2018). URL: https://thenewstack.io/debugging-microservices-lessons-from-google-facebook-lyft/.

[16] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. “Performance Monitoring and Root Cause Analysis for Cloud-Hosted Web Applications”. In: *Proceedings of the 26th International Conference on World Wide Web*. WWW ’17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 469–478. ISBN: 9781450349130. DOI: 10.1145/3038912.3052649. URL: https://doi.org/10.1145/3038912.3052649.

[17] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. “Deterministic replay of distributed java applications”. In: *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. IEEE. 2000, pp. 219–227.

[18] Aleksandra Kwiecień. “10 companies that implemented the microservice architecture and paved the way for others”. In: (2019). URL: https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/.

[19] Jonathan Mace. *End-to-End Tracing: Adoption and Use Cases*. Survey. Brown University, 2017.

[20] James Mickens, Jeremy Elson, and Jon Howell. "Mugshot: Deterministic Capture and Replay for Javascript Applications". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI'10. San Jose, California: USENIX Association, 2010, p. 11.

[21] Microsoft. "Monitoring a microservices architecture in Azure Kubernetes Service (AKS)". In: (2020). URL: https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring.

[22] A. J. Mirkin. "Containers checkpointing and live migration". In: 2010. URL: https://api.semanticscholar.org/CorpusID:115145450.

[23] Marco Mobilio et al. "Anomaly Detection As-a-Service". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2019), pp. 193–199.

[24] S. Nadgowda et al. "Voyager: Complete Container State Migration". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 2137–2142.

[25] Ravi Netravali and James Mickens. "Reverb: Speculative Debugging for Web Applications". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 428–440. ISBN: 9781450369732. DOI: 10.1145/3357223.3362733. URL: https://doi.org/10.1145/3357223.3362733.

[26] OpenZipkin. "Zipkin Architecture". In: (2020). URL: https://zipkin.io/pages/architecture.html.

[27] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture". In: *SIGSOFT Softw. Eng. Notes* 17.4 (Oct. 1992), pp. 40–52. ISSN: 0163-5948. DOI: 10.1145/141874.141884. URL: https://doi.org/10.1145/141874.141884.

[28] Anubha Sharma, Manoj Kumar, and Sonali Agarwal. "A Complete Survey on Software Architectural Styles and Patterns". In: *Procedia Computer Science* 70 (2015). Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems, pp. 16–28. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.10.019. URL: http://www.sciencedirect.com/science/article/pii/S187705091503183X.

[29] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[30] João Pedro Gomes Silva. "Debugging Microservices". In: (2019). URL: https://repositorio-aberto.up.pt/bitstream/10216/122187/2/350626.pdf.

[31] X. Zhou et al. "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study". In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2887384.

[32] Xiang Zhou et al. "Delta debugging microservice systems". In: *ASE*. 2018.

[33] Xiang Zhou et al. "Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, pp. 683–694. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338961. URL: http://doi.acm.org/10.1145/3338906.3338961.