DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Tail Based Sampling Framework for Distributed Tracing Using Stream Processing

G KIBRIA SHUVO

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Tail Based Sampling Framework for Distributed Tracing Using Stream Processing

G KIBRIA SHUVO

Master's Programme in Security and Cloud Computing
Date: 22.11.2021
Industry Supervisor: Miika Komu, DSc
KTH Supervisor: Peter Sjödin, PhD
KTH Examiner: Prof. Markus Hidell
Aalto University Examiner: Prof. Antti Ylä-Jääski
School of Electrical Engineering and Computer Science
Host company: Oy LM Ericsson Ab (LMF), Finland
Swedish title: Ramverk för svansbaserad provtagning för distribuerad spårning med hjälp av strömbearbetning

Abstract

In recent years, microservice architecture has surpassed monolithic architecture in popularity among developers by providing a flexible way of developing complex distributed applications. Whereas a monolithic application functions as a single indivisible unit, a microservices-based application comprises a collection of loosely coupled services that communicate with each other to fulfill the requirements of the application. Consequently, different services in a microservices-based application can be developed and deployed independently. However, this flexibility is achieved at the expense of reduced observability of microservices-based applications complicating the debugging of such applications. The reduction of observability can be compensated by performing distributed tracing in microservices-based applications. Distributed tracing refers to observing requests propagating through a distributed system to collect observability data that can aid in understanding the interactions among the services and pinpoint failures and performance issues in the system. OpenTelemetry, an open-source observability framework supported by Cloud Native Computing Foundation (CNCF), defines a standardized specification for generating observability data. Nevertheless, instrumenting an application with an observability framework incurs performance overhead. To tackle this deterioration of performance and to reduce the cost of persisting observability data, only a subset of the requests are typically traced by performing head-based or tail-based sampling. In this work, we present a tail-based sampling framework using stream processing techniques. The developed framework demonstrated promising performance in our experiments by saving approximately a third of memory-based storage compared to an OpenTelemetry tail-based sampling module. Moreover, being compliant with the OpenTelemetry specifications, our framework aligns well with the OpenTelemetry ecosystem.

Keywords

Distributed Tracing, Microservices, Observability, Monitoring

Sammanfattning

Under de senaste åren har mikrotjänstarkitektur överträffat monolitisk arkitektur i popularitet bland utvecklare genom att erbjuda ett flexibelt sätt att utveckla komplexa distribuerade tillämpningar. Medan en monolitisk tillämpning fungerar som en enda odelbar enhet, består en mikrotjänstbaserad tillämpning av en samling löst kopplade tjänster som kommunicerar med varandra för att uppfylla tillämpningens krav. Därför kan olika tjänster i en mikrotjänstbaserad tillämpning utvecklas och driftsättas oberoende av varandra. Denna flexibilitet uppnås dock på bekostnad av minskad observerbarhet för mikrotjänstbaserade tillämpningar, vilket försvårar felsökningen av sådana tillämpningar. Den minskade observerbarheten kan kompenseras genom att utföra distribuerad spårning i mikrotjänstbaserade tillämpningar. Distribuerad spårning innebär att man observerar förfrågningar som sprids genom ett distribuerat system för att samla in data om observerbarhet som kan hjälpa till att förstå interaktionerna mellan tjänsterna och lokalisera fel och prestandaproblem i systemet. OpenTelemetry, ett ramverk för observerbarhet med öppen källkod som stöds av Cloud Native Computing Foundation (CNCF), definierar en standardiserad specifikation för att generera observerbarhetsdata. Att instrumentera en tillämpning med ett ramverk för observerbarhet medför dock en överbelastning av prestanda. För att hantera denna försämring av prestanda och för att minska kostnaden för att bevara observerbarhetsdata spåras vanligtvis endast en delmängd av förfrågningarna genom att utföra s.k. “head-based sampling” eller “tail-based sampling”. I det här arbetet presenterar vi ett ramverk för tail-based sampling med hjälp av strömbehandlingsteknik. Den utvecklade ramen visade lovande prestanda i våra experiment genom att spara ungefär en tredjedel av den minnesbaserade lagringen jämfört med en OpenTelemetry-modul för tail-based sampling. Eftersom vårt ramverk är förenligt med OpenTelemetry-specifikationerna är det dessutom väl anpassat till OpenTelemetry-ekosystemet.

Nyckelord

Distribuerad spårning, mikrotjänster, observerbarhet, övervakning

Acknowledgements

To begin with, all praise to the Almighty God for his countless blessings. My appreciation goes to all who assisted and supported me throughout the whole process. I am extremely grateful to my examiner from KTH, Prof. Markus Hidell, and my supervisor from Aalto University, Prof. Antti Ylä-Jääski, for guiding my research work. Furthermore, I am deeply indebted to my advisor Miika Komu, DSc, for his constant support and valuable feedback throughout my thesis. The completion of this thesis would not have been possible without the support from Ericsson, Finland, and more specifically, my manager Jarno Kyykkä. Additionally, I would like to extend my sincere thanks to my colleagues, Juha Eskonen, Bruno Duarte Coscia, and Wäinö Kotilainen, for their valuable inputs. Last but not least, I must express my heartfelt gratitude to my family and friends for their constant support during the thesis.

Espoo, 22.11.2020

G Kibria Shuvo

With the support of the
Erasmus+ Programme
of the European Union

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Goals and Objectives . . . 3
- 1.3 Research Questions . . . 3
- 1.4 Benefits, Ethics and Sustainability . . . 3
- 1.5 Methodology . . . 4
- 1.6 Stakeholders . . . 4
- 1.7 Delimitations . . . 5
- 1.8 Outline . . . 5

2 Background and Related Work **6**
- 2.1 Virtual Machines and Containers . . . 7
- 2.2 Container Orchestration . . . 7
- 2.3 Monitoring and Observability . . . 8
- 2.4 Distributed Tracing . . . 9
  - 2.4.1 Components of a Trace . . . 9
  - 2.4.2 Request Correlation . . . 11
  - 2.4.3 Causality Preservation . . . 12
  - 2.4.4 Clock Skew Adjustment . . . 12
  - 2.4.5 Sampling . . . 13
- 2.5 Evolution of Distributed Tracing . . . 13
- 2.6 OpenTracing and OpenCensus . . . 17
- 2.7 OpenTelemetry . . . 17
  - 2.7.1 OpenTelemetry Collector . . . 20
- 2.8 Sampling Types . . . 20
  - 2.8.1 Head-based Sampling . . . 21
  - 2.8.2 Tail-based Sampling . . . 23

3 Design and Methodology 24
3.1 Ericsson Security Manager 24
3.2 Operational Requirements 25
3.2.1 Stream Processing 25
3.2.2 Flink and Kafka 26
3.3 Deployment Model 27
3.3.1 Dataflow Model 27
3.4 Algorithm 30
3.5 Feature Selection 31

4 Evaluation 33
4.1 Overhead of Tracing on ESM 33
4.2 Memory Requirements 34
4.2.1 Memory Consumption of OpenTelemetry Collector . . 35
4.2.2 Memory Consumption of Flink Cluster 36
4.3 Comparison 37

5 Discussion and Future Work 38
5.1 Overhead of Tracing on ESM 38
5.2 Capturing Insightful Traces 38
5.3 Benefits over OpenTelemetry Tail-based Sampling Processor . 39
5.4 Compatibility 40
5.5 Future Work 40

6 Conclusion 41

Bibliography 43

List of Figures

2.1 A hypothetical trace 10
2.2 A reference architecture of OpenTelemetry 18
2.3 An example pipeline in an OpenTelemetry collector 20
2.4 Comparison of head-based and tail-based sampling. In head-based sampling (a,b), the sampling decision is taken upfront. In tail-based sampling (c,d), the sampling decision is made in the tracing backend after the trace generation is complete [47] 21

3.1 Window types in stream processing 26
3.2 Interaction of the developed sampling framework with a standard OpenTelemetry deployment 28
3.3 Dataflow between the developed sampling framework and OpenTelemetry collector 28
3.4 Flowchart of the developed tail-based sampling algorithm . . . 31

4.1 Latency comparison of the scheduled tasks with tracing toggled 34
4.2 Latency increase of the scheduled tasks in percentage 35
4.3 Comparison of memory consumption of an OpenTelemetry collector instance in different scenarios: without tail-based sampling, while using the developed tail-based sampling framework, and while using OpenTelemetry tail-based sampling processor 36
4.4 Memory consumption of developed tail-based sampling framework running on a Flink cluster 37

Acronyms

AMQP Advanced Message Queuing Protocol 1

API Application Programming Interface 10, 14, 17, 22, 24

CAG Component Activity Graph 15

CNCF Cloud Native Computing Foundation 17

DAG Directed Acyclic Graph 19

DevOps Development and Operations 8

ESM Ericsson Security Manager 4, 24, 33, 38

ETW Event Tracing for Windows 14, 15

gRPC Google Remote Procedure Call 20

HTTP Hypertext Transfer Protocol 1, 15, 20

IaaS Infrastructure as a Service 6

IoT Internet of Things 24

LAF Least Appearances First 30, 38, 40

MLM Masked Language Modeling 16

NLP Natural Language Processing 16

OTel OpenTelemetry 17, 37

OTLP OpenTelemetry Protocol 20, 28, 30, 31, 40

PID	Proportional Integral Derivative 22
protobuf	Protocol Buffers 20, 28, 30, 31, 40
RPC	Remote Procedure Call 14
RPN	Reverse Polish Notation 14
SDK	Software Development Kit 17, 24
VM	Virtual Machine 6
VMM	Virtual Machine Monitor 7
W3C	World Wide Web Consortium 12, 17, 18

Chapter 1

Introduction

For many years, monolithic architecture has been the preferred choice for application development. According to monolithic architecture, applications are developed and deployed as single indivisible units. Over time, as a monolithic application becomes large, typically, different development teams focus on specific areas of the application. However, collaboration among specialized teams becomes increasingly difficult as they are bound to work on a single code base, thus making the teams tightly coupled with each other. Consequently, the development and deployment of a monolithic application require coordinated efforts from the development teams, which affect the pace of feature deliveries. To overcome the drawbacks mentioned above, nowadays, microservice architecture has been gaining popularity.

The flexible nature of microservice architecture has redefined the workflow of application development. According to microservice architecture, an application is a collection of loosely coupled services that communicate with each other in protocols such as Hypertext Transfer Protocol (HTTP) or Advanced Message Queuing Protocol (AMQP). As these services are weakly associated with each other, they can be developed, deployed, and scaled independently, enabling reliable and frequent delivery of complex distributed applications. However, these benefits are achieved at the expense of reduced observability of microservices-based applications. This reduction of observability stems from the fact that each individual microservice typically performs only a fraction of the work of a complete transaction, thus obfuscating the comprehensive overview of the transaction. Logs collected in a centralized logging server can aid in identifying the services participating in a transaction. However, the logging server cannot guarantee the orderly appearance of the logs. Consequently, in an erroneous transaction, finding the root cause of the error by correlating the logs can become extremely tedious. Distributed tracing can be employed to overcome the challenges mentioned above.

Distributed tracing refers to monitoring the requests propagating through a microservices-based application as part of a broader transaction. For this purpose, the first request of a transaction is tagged with a unique identifier that is propagated with all subsequent requests along the path of the transaction. As a result, all requests belonging to a transaction can be correlated using the unique identifier to generate a comprehensive overview of the transaction. However, instrumenting an application with distributed tracing introduces performance and storage overhead. To tackle this deterioration of performance and to reduce the cost of storing traces, only a subset of the requests are typically traced. This process of reducing the overhead and the cost of distributed tracing is called sampling. An efficient sampling method ensures the sampling of traces that provide crucial information for application performance evaluation and debugging in minimal performance and storage overheads. Therefore, sampling is pivotal to maximize the utility of distributed tracing.

The following sections in this chapter explain the motivation of the thesis, define the goals and objectives and outline the scope of this thesis.

1.1 Motivation

Introducing distributed tracing to an application is a trade-off between gaining better visibility into the system and the overhead of performance and storage. Sampling establishes a balance between these two compromises by reducing the performance overhead while capturing only the relevant traces to understand the system better. Therefore, establishing an effective sampling strategy is crucial for the success of distributed tracing. Additionally, the component performing the sampling of traces must be robust and capable of processing high volumes of incoming traces in minimal latency.

Sampling in distributed tracing can be categorized into two major categories, namely head-based sampling and tail-based sampling. In head-based sampling, the sampling decision is taken before the generation of the trace, whereas in tail-based sampling, the sampling decision is taken after generating the trace. Consequently, head-based sampling can effectively reduce the performance overhead of distributed tracing as the system does not experience any overhead when the sampling decision is negative. On the contrary, tail-based sampling excels at reducing the storage overhead as it produces more intelligent sampling decisions by evaluating the usefulness of each newly generated trace. Therefore, tail-based sampling ensures the sampling of traces that provide better insight into the system.

1.2 Goals and Objectives

This thesis aims to develop a framework to perform tail-based sampling in real-time for distributed tracing. The main objectives of the thesis are as follows:

- Develop a tail-based sampling framework that can be easily integrated into existing distributed tracing pipelines.
- Obtain better performance than the existing tail-based sampling component of the OpenTelemetry framework.

1.3 Research Questions

The aim of this work is to answer the following questions:

- How does the performance of an application change after introducing distributed tracing when compared to no tracing at all?
- How can we implement a tail-based sampling framework using stream processing?
- Which tail-based sampling method provides the best performance at scale?

1.4 Benefits, Ethics and Sustainability

The framework developed in this thesis can filter out frequently appearing traces, resulting in the sampling of less frequent but more informative traces relevant for debugging. The ability to capture only the relevant traces enables the users of this framework to maximize the utility of their limited storage capacity for storing observability data.

The recording and reporting of the performance metrics of the developed framework were achieved without any manipulation or fabrication. The framework does not collect any personal data; hence it is devoid of ethical issues related to improper handling of personal data. Plagiarism has been strictly avoided by properly citing existing works in this domain.

The framework is implemented using an established industry-standard open-source stream processing framework, which makes the proposed framework sustainable for the foreseeable future.

Furthermore, the framework brings a positive impact on the environment as it eliminates the necessity of acquiring additional storage space for storing observability data. Less storage requirement ensures better utilization of existing data centers resulting in reduced energy consumption. Consequently, the framework essentially aids in reducing the carbon footprint of the data centers leading to a sustainable future.

1.5 Methodology

First, we used a qualitative approach to gather requirements for developing a tail-based sampling framework for distributed tracing by surveying prior research works in this domain. Additionally, we explored the latest industry-standard distributed tracing solutions supporting tail-based sampling to establish a benchmark for performing quantitive analysis in the latter part of the thesis.

Based on the information gathered by the qualitative research, we designed the architecture of our tail-based sampling framework. The qualitative research findings revealed the stream processing paradigm as a suitable technology to perform tail-based sampling. Consequently, we implemented the tail-based sampling framework utilizing an industry-standard stream processing framework.

Finally, we took a quantitive approach to evaluate the performance of the developed tail-based sampling framework. We selected a set of metrics that are considered crucial for such tail-based sampling solutions and conducted experiments to compare the performance of the developed framework against an industry-standard solution. In light of the results from these experiments, we answered our research questions and provided our concluding remarks.

1.6 Stakeholders

This research was conducted at Ericsson, Finland, leveraging the tools and facilities provided by the company. The application used for the experiments is called Ericsson Security Manager (ESM) which is developed by Ericsson. The thesis was jointly supervised by Aalto University and KTH Royal Institute of Technology.

1.7 Delimitations

The tail-based sampling framework developed in this thesis uses an industry-standard open-source stream processing framework named Apache Flink. Additionally, the testbed for evaluating the performance of the developed framework involves another two open-source software, namely Apache Kafka and OpenTelemtry. Therefore, the performance of the framework is dictated by the performance of the software mentioned above.

1.8 Outline

The remaining of this thesis is organized as follows. Chapter 2 provides a background of distributed tracing and introduces relevant concepts; Chapter 3 details the design of the tail-based sampling framework developed for the thesis. Next, Chapter 4 evaluates the performance of the developed framework, while Chapter 5 discusses the benefits of using the developed framework. Finally, Chapter 6 provides concluding remarks.

Chapter 2

Background and Related Work

Microservice architecture refers to the decomposition of an application into a collection of loosely coupled services, each having a disjoint set of responsibilities[1]. This decomposition improves the maintainability and testability of the services. Additionally, it enables services to be deployed independently of each other[2]. The widespread adoption of microservice architecture can be attributed to the rapid development in the field of cloud computing (Infrastructure as a Service - IaaS)[3]. This advancement of cloud computing has been fostered by virtualization technology. Virtualization is the key component of cloud computing as it enables the sharing of computing resources, ensuring better resource utilization by running multiple operating systems on a single machine[4]. As virtualization technologies have matured, it has become easier than ever before to spin up a Virtual Machine (VM) to deploy applications. Eventually, containerization technologies such as *Docker* have enabled a single operating system to host multiple containers concurrently, providing utmost resource utilization[5]. Containerization enables services of an application to be deployed in the same operating system without the overhead of spinning up a new virtual machine.

In order to monitor and orchestrate these containerized services, technologies such as *Kubernetes* have stormed the industry. The orchestration services provided by Kubernetes have increased the reliability of microservices-based applications. These orchestration services include cluster management tasks such as service discovery, scheduling, and networking [6]. As a result, failures related to the network and underlying hardware are now abstracted away from the developers. Provided that the underlying abstraction is working correctly, it is now the responsibility of the developers to ensure the best performance of the application. Hence, developers require better observability into their systems more than ever to ensure that they have a shorter feedback loop that will allow them to rapidly make code changes to improve the quality of the services.

The following sections of this chapter introduce the notion of observability. In addition, it discusses how distributed tracing addresses this challenge in microservices-based applications.

2.1 Virtual Machines and Containers

System virtualization refers to using a software layer to imitate the behavior of a physical machine. This software layer is known as Hypervisor or Virtual Machine Monitor (VMM). The virtual environment created by this software layer is called a virtual machine. A virtual machine is capable of hosting an operating system independently, which makes it logically equivalent to a host machine [7]. Thus, hypervisors enable concurrent operation of multiple virtual machines inside a single physical machine facilitating efficient resource utilization. However, spinning up a new virtual machine to run an application incurs additional performance overhead. This overhead can be compensated by using containerization technologies.

A container is a self-sufficient package containing all the essential dependencies for an application to run. As a result, containerized applications are portable and readily executable on top of a container runtime. Container runtimes running on operating systems are responsible for container creation. Creating a container using a container runtime is analogous to starting a new process in isolation within the host operating system. This isolation is achieved by using Linux kernel-level features, namely *namespaces* and *cgroups (control groups)* [8]. Since containers share the kernel of the host operating system, the resource footprint of containers is lower than virtual machines. Consequently, containers enable efficient scaling of applications with reduced resource overhead which has further fostered the widespread adoption of microservice architecture [9], [10]. Docker is an industry-standard container engine that provides a complete ecosystem for managing Docker containers.

2.2 Container Orchestration

With the growing popularity of containerized microservices-based applications, the complexities of managing these deployments are increasing. While scaling out a service to accommodate a high volume of network traffic, the number of containers can rapidly become unmanageable for the Development and Operations (DevOps) team. Container orchestration solves this issue by automating all the tasks in a container lifecycle, including provisioning, deployment, scaling, networking, and load balancing. Container orchestration tools receive the desired state of the system in a configuration file and perform necessary operations to maintain the system in the desired state. Kubernetes is the industry-standard tool for container orchestration. Kubernetes can manage containers running in a cluster consisting of several host machines by performing cluster management jobs such as service discovery, scheduling, and networking [6].

2.3 Monitoring and Observability

According to *IEEE*, monitoring refers to supervising, recording, analyzing, or verifying the operations of a system[11]. Traditionally, monitoring is used for discovering predictable failures to determine the overall health of the system and generate alerts when necessary[12]. Monitoring can be of two types, *blackbox* monitoring and *whitebox* monitoring. Blackbox monitoring refers to observing a system from the outside without having any internal knowledge about it[13], [14]. Generally, blackbox monitoring involves querying the external characteristics of a service against several predefined scenarios[15]. In contrast, whitebox monitoring relies on knowledge about the internals of an application. Applications are instrumented with whitebox monitoring tools that emit telemetry data such as metrics, logs, events, and traces.

Regardless of the type of monitoring, it usually refers to reacting to failures once they occur. However, as applications are getting ever complex, the possible combinations of failures are increasing significantly. Thus, reacting after the occurrence of the incident is not viable anymore because the reaction time after a failure is becoming longer with the increasing complexity of the services. Therefore, it is best to detect symptoms before the incident. Nowadays, DevOps teams need to be proactive even for unpredictable failures and try to anticipate them beforehand. Precisely on this scenario is where observability comes into the picture.

In terms of control theory, observability denotes the ability to predict the internal state of a system based on its output[16]. In terms of distributed systems, observability refers to the ability to extract useful information from the telemetry data to understand the internals of a system. Thus, it is possible to uncover underlying asymptomatic systemic issues in an observable system. Monitoring can provide the DevOps team with the data that presents an overview of the system and enable the generation of alerts when required. In contrast, observability is the extraction of useful information from the collected telemetry data that can be useful for debugging, profiling, and dependency analysis. Therefore, observability is a superset of monitoring.

The three pillars of observability in distributed systems are logs, metrics, and traces. First, logs represent records of immutable discrete past events with timestamps in plain text, structured or binary format. Next, metrics are numeric values of selected data points of a system that are measured over a time interval. Metrics provide valuable insights into the internal state of a system at any given point in time. Finally, traces are reconstructions of causally related events of the transactions in a distributed system. Traces provide an overview of end-to-end request flow through a distributed system[12]. In the following sections of this thesis, we will focus on traces in particular.

2.4 Distributed Tracing

Distributed tracing refers to monitoring the requests of a transaction taking place in a microservices-based application. The requests are monitored by attaching relevant contextual metadata along with them during their execution. Thus, distributed tracing allows the reconstruction of the events during the transaction while maintaining the causal relationship among them. A collection of such causally related events are referred to as a trace. Distributed tracing is different from conventional code profilers and host-level tracing tools such as *DTrace*[17]. Whereas traditional code profilers can monitor code execution within a single host, distributed tracing focuses on monitoring applications running on different hosts[3]. Consequently, distributed tracing is pivotal for monitoring microservices-based modern cloud-native applications distributed among multiple hosts.

2.4.1 Components of a Trace

The terminologies which are used to describe modern distributed tracing concepts can be linked back to systems such as *Dapper*[18], *Magpie*[19] and *X-Trace*[20]. A *trace* is essentially a collection of spans that are causally related to each other. A *span* represents a unit of work performed by a service taking part in a broader transaction. Each span has a duration associated with it which represents the time taken to perform that specific operation. Additionally, it is possible to add extra key-value attributes to spans. All spans within a trace share a parent-child relationship between them. The span which does not have

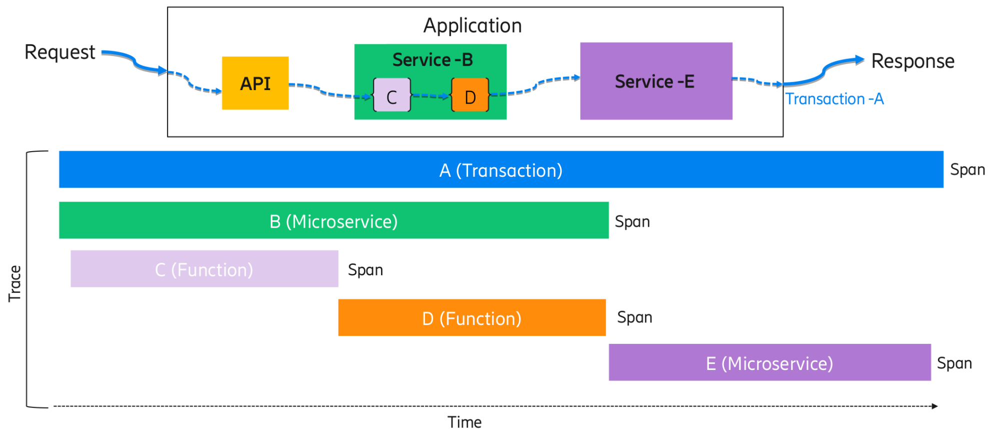


Figure 2.1: A hypothetical trace

any parent span is known as the root span. The duration of the root span denotes the end-to-end latency of a complete transaction. The section of code that is instrumented to generate new spans is called a tracing point. The instrumentation is performed by a tracing library that injects contextual metadata into the requests, enabling the formation of the causal relationship between spans.

Figure 2.1 illustrates the composition of a hypothetical trace. In this figure, a microservices-based application has two services, namely Service-B and Service-E, instrumented using a tracing library. As a request hits an Application Programming Interface (API) endpoint of the application, the tracing system starts a span named A, which will denote the end-to-end latency of the complete transaction. As time progresses and the request hits Service-B, the tracing system starts a new span named span B representing the work performed in Service-B. Inside Service-B, two functions, namely C and D, are also instrumented to generate spans. As a result, after each of these functions completes its execution, the tracing system generates two new spans: span C and span D. After generating these new spans, the tracing system marks the end of span B. Eventually, the request hits Service-E, which results in the generation of a new span named span E. Finally, as the system finishes execution of the request and generates a response, the tracing system marks the end of the span A, which concludes the generation of the trace for this transaction.

2.4.2 Request Correlation

When compared with other profiling tools, distributed tracing has the distinctive ability to correlate the newly generated profiling data among each other correctly. The task of request correlation can be performed in multiple ways, such as *blackbox* inference, *schema-based* inference, and *metadata-based* context propagation[3].

Blackbox inference aims at performing request correlation without requiring any modification to the system being monitored. Instead, this technique relies on statistical analysis or machine learning to perform request correlation. One example of a such system is the *Mystery Machine*[21] developed at Facebook, which uses big-data techniques to process the logs generated by large-scale internet services. Mystery Machine applies big-data techniques on these logs to construct a model of request execution. Blackbox inference techniques might initially seem appealing as they do not require any modification to the system. However, blackbox inference techniques rely on massive computation to infer request correlation, making them expensive and slower than both schema-based and metadata-based request correlation.

Schema-based inference techniques also do not require modification to the applications; nonetheless, applications are required to have their own event schema defined. One such system is *Magpie*[19], which can extract the control flow and resource consumption of the requests of a system using an application-specific event schema[22]. However, this approach fails to keep up with modern large-scale distributed systems as the creation of application-specific event schemas cannot be automated.

Finally, the metadata-based context propagation method relies on annotating the executions within a request using a global execution identifier that allows the tracing system to reconstruct the complete execution of the request. The application requires instrumentation to annotate the requests; these annotations inject the execution contexts into the requests as they propagate through the system. Such context propagation can be performed in two methods, namely *in-process* and *inter-process* propagation. The in-process propagation method makes the metadata available within the same process that is dealing with thread switching and other asynchronous behavior. Conversely, inter-process context propagation deals with transferring metadata across different services over network calls.

Modern distributed tracing systems utilize the metadata-based context propagation method for performing request correlation because it is more flexible and can generate traces precisely and faster than the alternative methods[3].

World Wide Web Consortium (W3C) recommends a standard for context propagation named *Trace Context*[23] to ensure interoperability between different tracing systems.

2.4.3 Causality Preservation

The next challenge in generating distributed traces is to preserve the causality relation between the generated spans. Request correlation is only sufficient for grouping all the spans of a particular request. However, reconstructing a graph from these spans while maintaining all the causally related activities requires additional information. The reconstruction can be achieved by following Lamport's *happens-before* relation[24] denoted by " $\longrightarrow$ ". According to this principle, three events: a, b, and c have the relation $\longrightarrow$ if they satisfy the following conditions:

1. If event a and event b occurs in the same process then, $a \longrightarrow b$ (a happens-before b), provided that event a appeared before event b.

2. If event a represents sending a message and event b represents the reception of the same message, then $a \longrightarrow b$ (a happens-before b).

3. $a \longrightarrow b$ and $b \longrightarrow c$ implies that $a \longrightarrow c$ (a happens-before c).

However, depending solely on Lamport's happens-before principle is insufficient because it might capture false causality relations as it only relies on the event time to determine causality. To mitigate this issue, modern tracing systems, such as *Dapper*[18] and *X-Trace*[20], use dynamic metadata for attaching an execution identifier for each of the newly generated traces. This execution identifier is further propagated towards the next tracing point, where this inbound execution identifier is considered as tracing data. This next tracing point stores the incoming execution identifier as its parent and injects a new execution identifier to propagate the causality relation towards its children.

2.4.4 Clock Skew Adjustment

Generally, timestamps captured within the same process are expected to be aligned with each other. However, timestamps from different processes in the same host may differ due to many factors, such as the programming language and the libraries used to generate the timestamps. As distributed tracing deals with an even more challenging scenario, that is, monitoring request executions in different hosts, it is inevitable that distributed tracing has to deal with clock skew. Distributed tracing reconciles the clock skews by realigning the spans according to the causality relations. For example, if a child span appears before a parent span in the trace view, it is adjusted by adding an offset to the child span until the complete child span falls within the time range of the parent span. However, this adjustment is based on applying heuristics. Therefore, it is not entirely accurate.

2.4.5 Sampling

One of the unwanted side-effects of introducing distributed tracing to any application is the performance overhead of generating tracing data. Moreover, additional storage is necessary to persist the generated traces for further analysis. Sampling tackles both of the problems by reducing the amount of tracing data being generated and stored. Through sampling, the developers can find a reasonable rate at which the business logic is not overwhelmed by the generation of tracing data. At the same time, the budget for storing traces remains affordable. As sampling is the decisive factor that dictates the overall success of adopting distributed tracing in any organization, it is discussed in more detail in Section 2.8.

2.5 Evolution of Distributed Tracing

Historically, tracing is a time-consuming and manual task. All popular operating systems, including Linux and Windows, support numerous tools for performing tasks such as tracing, debugging, and performance evaluation. The tracing tools available for Linux can be divided into two major categories, namely *ptrace-based* application tracing and *kernel-based* tracing[25]. *Ptrace* is a system-call of Linux, which is at the core of popular tools such as *gdb*, *strace* and *ltrace* [26]. However, ptrace-based debuggers and tracing tools exhibit poor performance due to overheads of context switching [27], which can be mitigated by using kernel-based tools.

Kernel-based tracing can be performed in two methods: *kernel-module* approach and *interpreter-based* approach. The kernel-module based tracing tools include *kprobes*, *utrace*, *uprobes*, and *SystemTap*. Kprobes was released in the Linux kernel version (2.6.9-rc2), and it works by dynamically inserting probes into a running kernel and replacing the program text with a breakpoint instruction. Upon hitting a probepoint, the kprobes infrastructure takes control and executes a user-defined handler in the context of the kernel [27]. Utrace can trace user-space applications by placing tracepoints at strategic locations in the kernel code. Once the tracepoints are activated, they invoke the pre-registered utrace clients in kernel mode. Uprobes works by adding a user-defined kernel module having probepoints along with handlers to be executed upon a probepoint activation[26]. SystemTap acts as a wrapper for kernel-based instrumentation tools such as kprobes. It simplifies the usage of kprobes by accepting a script written in a language similar to AWK scripting language for making API calls to kprobes[26], [28].

Interpreter-based tracing tools in Linux include *DProbes* and *DTrace*[25]. DProbes allows the dynamic insertion of probepoints into code modules executing in either user and kernel space. When a probepoint is activated, DProbes runs a user-defined probe-handler written in a Reverse Polish Notation (RPN) based assembly language, which gets interpreted in the kernel[25], [29]. DTrace can instrument machine code, enabling it to monitor machine codes translated by various programming language interpreters. Thus, DTrace can monitor programs written in any interpreted language. The language for writing probe handlers for DTrace is written in D language, which shares similarities to C and AWK programming languages[30].

Similar to Linux, Windows also provides a low-overhead, non-blocking kernel-level tracing facility named *Event Tracing for Windows (ETW)*. ETW allows tracing various application and operating system events via a Win32 API function. An event record in ETW consists of an event type, and other data fields specific to the event. Furthermore, ETW accepts additional data fields for specifying user-defined data[31] for debugging purposes.

However, the tracing systems discussed above only work within a single host, which is insufficient for tracing microservices-based applications as these applications are often distributed among multiple hosts. Tracing for distributed systems can be classified into two categories, namely *blackbox* and *whitebox* tracing. *DPM*[32] is among one of the initial projects in the field of blackbox tracing. DPM relies on kernel instrumentation to determine causality between requests. *Project 5* approaches the blackbox tracing problem for local-area distributed systems by providing two different algorithms: one uses time series analysis of Remote Procedure Call (RPC) messages, and the other relies on a convolution algorithm based on signal-processing techniques to identify the causal path of requests[13]. The successor of Project 5 is *Wide-Area Project 5 (WAP5)*[14], in which a new message-linking algorithm performs causal-path inference to introduce performance tracing support for wide-area systems. *E2Eprof*[33] uses a pathmap algorithm similar to the convolution algorithm used in Project 5. However, due to using a compact trace representation, E2Eprof is more lightweight than Project 5 and can perform online performance diagnosis. *BorderPatrol* [34] unbundles concurrent requests at the protocol level in order to precisely monitor the output of each request on a per-event basis. *PreciseTracer*[28] can trace multi-tier blackbox services using request tracing leveraging the SystemTap tool without requiring any application-specific knowledge. Additionally, PreciseTracer can generate a Component Activity Graph (CAG) representing the causal path of requests. *Magpie*[19], developed in Microsoft research, works by instrumenting a distributed system with ETW to generate named events with timestamps. Association of events with the requests and behavioral clustering of requests to detect anomalies are performed offline. *Pinpoint*[35], [36] follows a similar design and philosophy as Magpie. However, instead of focusing on performance analysis, Pinpoint targets fault detection within distributed systems. *CLUE*[37] uses a data-centric approach to evaluate the performance of cloud-based systems by monitoring their interactions with the underlying hardware. CLUE leverages kernel-level event sequencing to identify performance issues in cloud applications.

PSpec[38] is a whitebox tracing system that allows developers to write assertions about the system behavior, which are checked against the collected monitoring data from a pre-instrumented application. PSpec only aims at detecting performance bottlenecks, and it cannot determine causal paths or application structure. This drawback has been addressed in *Pip* [39], which aims at detecting bugs in a distributed system by comparing the perceived system behavior along with expected system behavior. Pip allows developers to instrument their system to collect actual system behavior. Additionally, they can define their expectations of timing, protocols, and resource consumption in a declarative language. Finally, they can use query and visualization tools provided by Pip to understand and debug the system. Analogous to Pip, *Paradyn*[40] allows the developers to write their expectations of the system using Paradyn Configuration Language (PCL). However, similar to PSpec, Paradyn cannot detect causal path or the system structure. *WebMon*[41] can monitor the performance of web services by instrumenting applications to use HTTP cookies for performing request correlation. WebMon uses a sensor-collector architecture, where sensors generate events by injecting special correlators into HTTP cookies and forwarding them to the collector. The collector enables further processing of the data by making it available in a persistent storage. *Stardust*[42] provides an infrastructure to collect end-to-end traces of distributed storage systems. Stardust uses a component named activity tracking infrastructure (ATI), which monitors and measures CPU demand, buffer cache usage, network demand, and disk demand of every client request. Finally, it sends the activities to a querying infrastructure for future analysis.

The root of modern distributed tracing systems can be traced back to systems such as *X-Trace*, *Dapper*, and *Canopy*. X-Trace[20] generates a unique task identifier for each request and injects additional metadata along with the request. These are propagated along with all recursive requests, resulting in a task tree representing all the network calls originating from the initial request. However, X-Trace requires the underlying networking stack to be modified to enable the propagation of the metadata. Dapper[18] was initially developed in Google to monitor their web-search workloads. Most of the terminologies used in modern tracing systems were first defined in Dapper. Dapper follows a similar workflow as X-Trace and propagates request contexts across all the services. According to Dapper, each unit of work along the path of a transaction is called a span, and a collection of spans constitute a trace. Additionally, Dapper allows annotations of the spans, enabling the developers to add additional information to the spans. Canopy[43], developed by Facebook, extends the core concept of systems such as Dapper and XTrace. However, Canopy primarily emphasizes solving the challenges faced during scaling distributed tracing systems. Canopy addresses the challenges by separating instrumentation from trace models, providing a pipeline for extracting features from traces, and presenting different views of the same tracing data for different use cases.

Modern open-source distributed tracing systems include *Apache SkyWalking*[44], *Zipkin*[45], and *Jaeger*[46]. Zipkin and Jaeger were initially developed in-house in Twitter and Uber, but later, these tracing systems were open-sourced. Skywalking was an open-source system from its inception, and it gained attention once it was accepted as an incubator project in Apache foundation.

As discussed in Section 2.4.5, sampling is the deciding factor in ensuring the effectiveness of distributed tracing. As a result, researchers in this area attempt to evaluate the effectiveness of different sampling algorithms that perform sampling during distributed trace generation. *Sifter*[47] uses a Natural Language Processing (NLP) technique called Masked Language Modeling (MLM) [1] to perform sampling of traces. Sifter maintains an online machine learning [2] model of the common execution paths of a distributed application. Using this model, Sifter can distinguish the edge cases from the common execution paths and sample the traces representing the edge cases. Las-Casas *et el.*[48] provide a clustering-based method to determine common execution paths

[1]In MLM, a language model has access to only parts of a sequence of input tokens where the rest of the tokens are replaced with a special token called a mask. The model attempts to reconstruct the original sequence of tokens by predicting the masked tokens.

[2]Online machine learning is used to train machine learning models using streaming data where the model is updated on the arrival of new data points

of a distributed application. Their method is biased towards sampling execution graphs that do not fall into the common clusters, resulting in the sampling of graphs representing infrequent execution patterns. Mace *et el.*[49] use graph kernels [3] to evaluate trace similarity. In this method, graph representations of newly generated traces are momentarily stored in memory and evaluated using graph kernels to calculate similarity with existing graphs in a cluster. Sampling decisions are made based on the similarity scores, and graphs having lower similarity scores are prioritized.

2.6 OpenTracing and OpenCensus

OpenTracing was initiated with a vision of becoming an open-source, vendor-neutral standard for distributed tracing. It provided a vendor-neutral API for instrumenting applications along with instrumentation libraries the for most popular frameworks. In 2016, it was accepted as a project under the Cloud Native Computing Foundation (CNCF)[50], which increased the acceptance of the project in the developer community devoted to distributed tracing. A few years later, in 2018, Google open-sourced their distributed tracing and metrics generation framework named *Census* as *OpenCensus*[51]. Both OpenTracing and OpenCensus had the same goal of standardizing the domain of distributed tracing. However, due to different architectural designs, the projects eventually became a barrier to the standardization of distributed tracing. To mitigate this issue, leadership from both projects decided to collaborate and created a single project under CNCF called *OpenTelemetry* in 2019[52].

2.7 OpenTelemetry

OpenTelemetry (OTel), an open-source framework for observability, is the merger of two previously popular open-source projects, namely OpenTracing and OpenCensus. OpenTelemetry is a collection of vendor-agnostic API, Software Development Kit (SDK), and tools for generating, instrumenting, and collecting telemetry information such as traces, metrics, and logs from modern cloud-native applications[53]. OpenTelemetry has well-defined specifications for all telemetry data types[54]. OpenTelemetry also outlines the process of transporting the generated telemetry data between different components of a tracing infrastructure. OpenTelemetry is W3C *Trace Context*[23] compliant,

[3]A graph kernel is a function that reports the similarity of two graphs via a numeric value.

which makes it compatible with any monitoring system following the W3C Trace Context recommendation.

Figure 2.2 illustrates a typical OpenTelemetry deployment. In the diagram, the services of an application are instrumented using the OpenTelemetry library, which performs the task of generating telemetry data. Once generated, the telemetry data are sent to an OpenTelemetry collector instance called OpenTelemetry *agent*. OpenTelemetry agents can be deployed as a sidecar container [4] of each service of an application or as a Kubernetes DaemonSet [5], which ensures that an OpenTelemetry agent daemon runs inside every host of a Kubernetes cluster. The agent relieves the services from the burden of batching, retrying, encrypting, and compressing of tracing data. The agent sends the telemetry data to an OpenTelemetry *collector*, which then processes and exports the data to a storage backend. Finally, a trace querier such as *Jaeger-UI* or *Zipkin* can query the traces from the storage backend and visualize them to the developers.

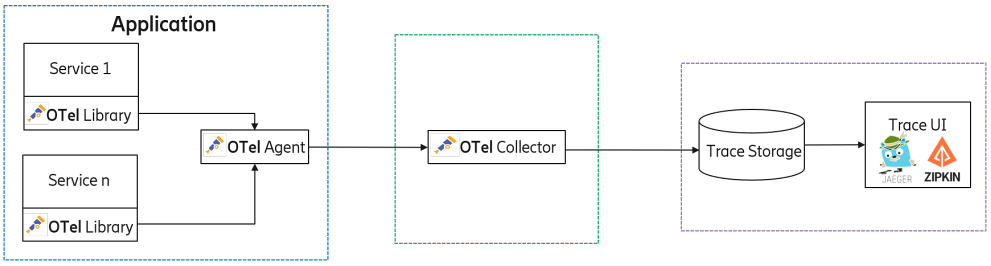


Figure 2.2: A reference architecture of OpenTelemetry

OpenTelemetry defines semantic conventions for specifying keys and values for the telemetry data generated by the applications. The following sections describe the semantic conventions for OpenTelemetry tracing data.

[4]A sidecar container enhances the functionalities of an application container and resides in the same atomic container group as the application container[55].

[5]A DaemonSet is a Kubernetes construct that ensures that a copy of a pod(Kubernetes construct representing an atomic container group) always runs in the selected nodes of a Kubernetes cluster.

Spans:

According to OpenTelemetry specifications[54], a span represents a unique operation within a transaction. A span consists of the following states:

- Operation name
- Timestamps (Start, Finish)
- Attributes list (Key-value pair)
- Event set (Zero or more)
- Parent span identifier
- Links to other causally-related spans (Zero or more)
- SpanContext information

SpanContext:

SpanContext relates a span with a trace using tracing identifiers. SpanContext must be propagated to child spans and across process boundaries to ensure the correct reconstruction of a trace. The following fields are used to establish SpanContext.

- Trace identifier (TraceId)
- Span identifier (SpanId)
- Trace flags (TraceFlags)
- Trace state (Tracestate)

Traces:

A trace is essentially a collection of spans. According to OpenTelemetry specification [54] a trace is a Directed Acyclic Graph (DAG) of spans in which the edges between the spans represent parent-child relationships. However, the most popular distributed trace visualization tools represent traces using charts similar to *Gantt* charts with a time axis, as shown in Figure 2.1.

2.7.1 OpenTelemetry Collector

OpenTelemetry collector offers a vendor-neutral way to collect, process, and export telemetry data. Being a vendor-agnostic tool, it can receive telemetry data from any sources irrespective of the inbound data format, then process and finally export the data to any commercial backend. The whole process of receiving, processing, and exporting telemetry data is performed in a pipeline, and multiple pipelines can exist in a single OpenTelemetry collector. A typical OpenTelemetry pipeline consists of a number of *receivers*, followed by a chain of *processors* and completed by a number of *exporters*. Receivers receive telemetry data in a specified format and translate the data into an internal format before passing it to processors or exporters. A single receiver can send the same data to multiple pipelines using a fan-out connector. Each processor receives the tracing data strictly from a single receiver or a preceding processor and sends it to the following processor or an exporter. Processors can transform the tracing data by performing operations such as batching, filtering, and tail-based sampling. Finally, exporters convert the tracing data into the specified format and forward the data to telemetry data storage backends. Figure 2.3 illustrates an example pipeline in an OpenTelemetry collector. Communication between different telemetry sources, intermediate nodes, and telemetry backend is specified by OpenTelemetry Protocol (OTLP). OTLP specifies the encoding, transport, and delivery of telemetry data. OTLP is based on Google Remote Procedure Call (gRPC) and HTTP/1.1 and uses Protocol Buffers (protobuf) schemas for handling payloads.

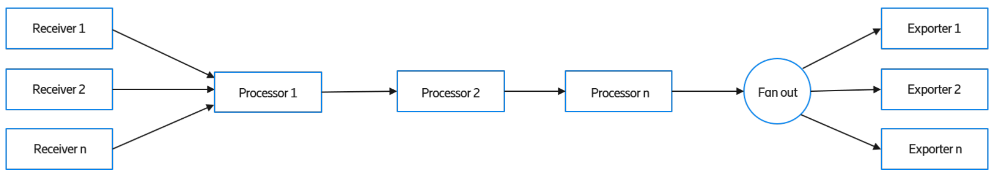
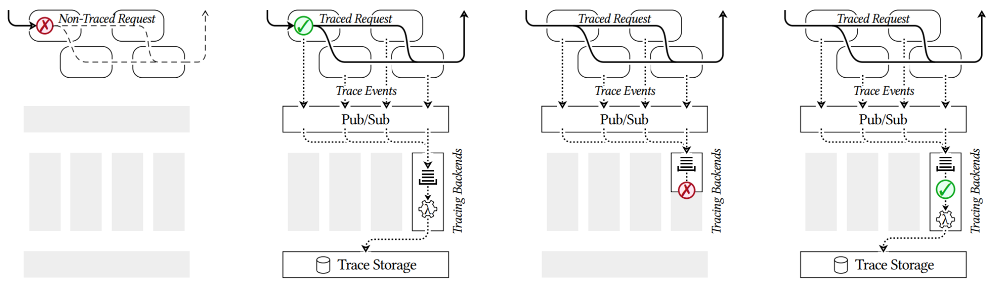


Figure 2.3: An example pipeline in an OpenTelemetry collector

2.8 Sampling Types

Sampling in distributed tracing is the process of generating a selective amount of traces in order to reduce the performance and storage overhead. Applications serving a heavy amount of traffic might exhibit poor performance due to the lack of proper sampling. An example is the *Dapper* tracing system from Google, which initially introduced 1.5% throughput and 16% latency overhead for web search workloads without any sampling. By reducing the sampling rate to 0.01% of the traces, both throughput and latency overhead were reduced to 0.06% and 0.20% , respectively[18]. The reduction of throughput and latency demonstrates the importance of sampling in distributed tracing.

There are two main types of sampling in distributed tracing, namely *head-based* sampling and *tail-based* sampling. Figure 2.4 illustrates both head-based and tail-based sampling in action. In head-based sampling, the sampling decision is taken upfront before generating any spans. Therefore, in head-based sampling, the system experiences overhead due to tracing only if the sampling decision is positive. Conversely, in tail-based sampling, the traces are generated before taking the sampling decision. The generated traces are sent to a tracing backend where the sampling decision is taken. As a result, tail-based sampling can sample more informative traces and provide better insight into the system as the newly generated traces can be taken into account to produce a biased sampling decision. Both of these sampling types are detailed below.



(a) Head-based, not sampled. (b) Head-based, sampled. (c) Tail-based, not sampled. (d) Tail-based, sampled.

Figure 2.4: Comparison of head-based and tail-based sampling. In head-based sampling (a,b), the sampling decision is taken upfront. In tail-based sampling (c,d), the sampling decision is made in the tracing backend after the trace generation is complete [47]

2.8.1 Head-based Sampling

In head-based sampling, the sampling decision is taken upfront before generating any spans. Tracing libraries take a sampling decision once for every trace, and the decision is propagated to all tracepoints in the system for maintaining consistency. Head-based sampling ensures that either all or none of the spans of a particular trace are generated. Hence, head-based sampling reduces the overhead when the sampling decision is negative. Head-based sampling is employed by tracing systems such as *Dapper*[18] of Google and *Canopy*[43] of Facebook. Moreover, all modern distributed tracing systems support head-based sampling. The sampling decision of head-based sampling can be obtained based on several methods, as discussed below.

Probabilistic Sampling:

Probabilistic sampling is the most common and popular sampling technique used by many modern tracing systems today for performing head-based sampling[3]. In probabilistic sampling, the sampling decision is taken uniformly at random. Popular tracing systems, such as *Jaeger*[46] developed by Uber, and *Spring Cloud Sleuth*[56], use probabilistic sampling as default. Although probabilistic sampling is the simplest method to perform head-based sampling, this method cannot control the selection of traces during sampling. As a result, the sampled traces might not reflect the overall condition of the system.

Rate Limiting Sampling:

Rate limiting sampling is another simple sampling technique for performing the head-based sampling. In a rate limiting sampling scenario, only a fixed number of traces are allowed to be generated in a unit of time. The rate limiting is enforced by using a rate limiting algorithm such as the *leaky bucket* algorithm. Rate limiting sampling is beneficial for services having fluctuating traffic patterns, as probabilistic sampling is not useful in such cases.

Adaptive Sampling:

In a microservices-based application, each service experiences a different load due to serving a different volume of traffic. However, probabilistic sampling and rate-limiting sampling do not take this varying load into account. As a result, all API endpoints get equal priority for sampling. Adaptive sampling aims at solving this problem by dynamically adjusting the sampling parameters during runtime. This concept was first introduced in the *Dapper* tracing system of Google. In this method, Dapper used to adjust the sampling probability over time across all the services based on the traffic load[18]. *Jaeger* tracing developed by Uber provides a similar but more sophisticated implementation of adaptive sampling using the Proportional Integral Derivative (PID) controller concept. In their implementation, the adaptive sampling infrastructure calculates the desired sampling probability for each service during runtime. A feedback loop pushes the newly calculated sampling probabilities back to the tracing libraries. Afterwards, the tracing libraries use the updated sampling probabilities during the sampling traces of different services[3].

2.8.2 Tail-based Sampling

Tail-based sampling takes the sampling decision after generating the trace. As a result, tail-based sampling does not play any role in reducing the performance overhead of the application. Instead, tail-based sampling is useful for sampling traces that provide crucial information about the overall status of the system. In tail-based sampling, the newly generated trace itself can be factored into the sampling decision. Thus, tail-based allows biased sampling (i.e., storing only the useful traces, discarding others). Consequently, tail-based sampling enables the storing of qualitatively better traces without exceeding the storage budget[47].

Chapter 3

Design and Methodology

The aim of this thesis is twofold. First, to measure the overhead of distributed tracing on Ericsson Security Manager (ESM) by instrumenting ESM with OpenTelemetry API and SDK. Second, to develop a novel tail-based sampling framework compliant with the OpenTelemetry specifications. This chapter introduces ESM and details the design architecture of the developed tail-based sampling framework. Furthermore, this chapter explains the design choices made during the development of the tail-based sampling framework.

3.1 Ericsson Security Manager

ESM is a microservices-based software for automating network configuration and management tasks in telecommunication, Internet of Things (IoT), and private networks. It can enforce security policies on network nodes and automate the process of compliance monitoring of network nodes. Additionally, ESM can detect probable threats in any of its managed networks using predefined threat detection logic and take necessary actions to mitigate the impact of the threats. As a microservices-based application, it is imperative for ESM to employ distributed tracing to increase the observability of the application. However, it is also crucial to verify that the overhead of distributed tracing does not overwhelm the business logic of ESM. For this purpose, we instrument a selected set of services in ESM using OpenTelemetry and measure and evaluate the impact of introducing distributed tracing. Insights from this evaluation will be taken into consideration when instrumenting more services of ESM in the future.

3.2 Operational Requirements

The sampling of distributed traces is a task to be inherently accomplished online. Sampling systems need to process a continuous stream of incoming spans and produce the sampling decisions in real-time. Hence, the sampling framework has to be developed using technologies that are robust and capable of dealing with large-scale streaming data efficiently. Stream processing is a paradigm that satisfies the requirements of developing such a tail-based sampling framework.

Another major motivation behind performing tail-based sampling is to collect insightful traces which provide crucial information about the performance and behavior of the application. The usefulness of the traces can be determined by capturing and modeling the common executions path of an application so that the framework can prioritize the uncommon executions paths representing more insightful traces. Therefore, an additional requirement for the underlying stream processing framework is to support stateful processing of incoming data streams to enable the modeling of common executions paths. These requirements played a vital role in selecting the underlying stream processing framework to be used to build our tail-based sampling framework.

3.2.1 Stream Processing

Stream processing originates from big data analysis for handling unbounded continuous data. Stream processors ingest a continuous stream of events and trigger a predefined action for each event. This method differs from the traditional processing of data stored in data at rest [1] infrastructures; instead of waiting for all of the events to arrive, stream processing aggregates the events over time to produce real-time outputs. The core principle in stream processing is to partition infinite data streams into small finite windows and apply computation on them to enable real-time outputs with minimal latency.

There are three main types of windows in stream processing, namely *tumbling* or *fixed* windows, *sliding* windows, and *session* windows, as shown in Figure 3.1. Tumbling windows refer to slicing up the incoming data streams into time intervals of fixed length without overlap. Sliding windows are fixed-length windows that are spawned after a fixed period of time. Finally, session windows are created upon arrival of events and terminated after a fixed time of inactivity, resulting in a timeout.

Stateful stream processing is a special case of stream processing where the stream processor maintains states containing information about the events

[1]Data at rest refers to storing data in any digital format in a persistent storage device

that appeared previously. Maintaining past events information is called state management, and it can be performed by storing the state in memory or external storage. In memory, state management is only suitable for processing jobs with moderately sized states. For processing jobs having significant state storage requirements, external key-value stores are more appropriate.

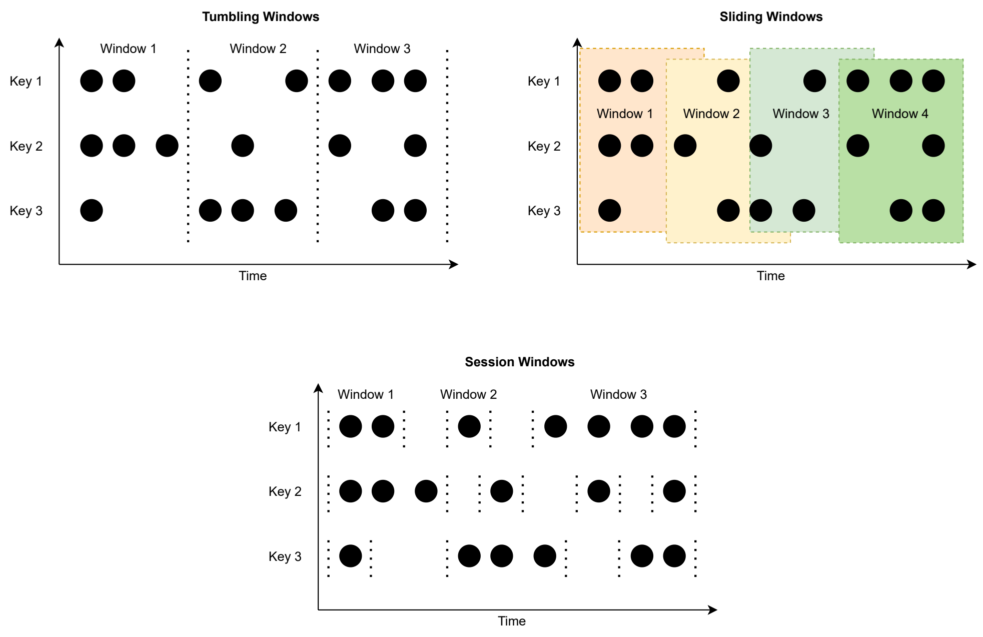


Figure 3.1: Window types in stream processing

3.2.2 Flink and Kafka

Apache Flink[57] is a popular open-source framework written in Java supporting both batch and stream processing. Flink is the first open-source stream processing framework to introduce native support for processing unbounded data based on event time [58] using watermarks. [2] Watermarks denote the progression of event time in Flink, allowing Flink to identify and handle out-of-order data. On the contrary, alternatives such as Apache Spark perform micro-batch operations to imitate stream processing. Flink is used in conjunction with durable message queues, such as *Apache Kafka*[61], to achieve

[2]Watermark is a monotonically increasing timestamp of the latest data point that has been processed by a stream processing engine[59]. A data point having a lower timestamp than the watermark is considered as late[60].

high throughput and low latency, ensuring exactly-once delivery guarantees. Due to these aforementioned benefits, we developed our tail-based sampling framework using Apache Flink.

Furthermore, we also used an open-source durable message bus called Apache Kafka because it is highly scalable [62]. Kafka enables us to achieve high throughput ingestion of spans which is pivotal for ensuring sampling decisions within minimal latency in services with high-volume traffic.

3.3 Deployment Model

One of the main requirements for our tail-based sampling framework is to make it compatible with existing OpenTelemetry deployments. Since we are developing a tail-based sampling framework, deploying our framework along with an OpenTelemetry collector instance is logical to enable immediate access to the newly generated spans. Moreover, we decouple our framework from the application being traced by deploying the framework with OpenTelemetry collectors. Therefore, no further modifications are required to a previously instrumented application for using our sampling framework. Furthermore, the developed tail-sampling framework accepts spans in accordance with OpenTelemetry specifications, enabling smooth integration with any OpenTelemetry collector instance.

Figure 3.2 shows the integration of the developed tail-based sampling framework with an existing OpenTelemetry deployment. In this figure, the OpenTelemetry collector forwards the stream of spans to the developed framework. The framework starts to process incoming spans and returns tail-based sampling output to the OpenTelemetry collector instance. Upon receiving the sampling results, the OpenTelemetry collector finally exports sampled spans to the trace storage. The following section describes the dataflow model of our tail-based sampling framework.

3.3.1 Dataflow Model

As discussed in Section 2.7.1, an OpenTelemetry collector can have multiple pipelines to process incoming traces. Our tail-based sampling framework requires the collector instance to have two pipelines, as shown in Figure 3.3. Code snippet 1 shows the configuration of an OpenTelemetry collector for this dataflow model. The configuration has two pipelines, namely *traces/input* and *traces/output*. The *traces/input* pipeline receives spans via an *otlp* receiver and passes them to the *batch* processor. Using a batch processor is optional

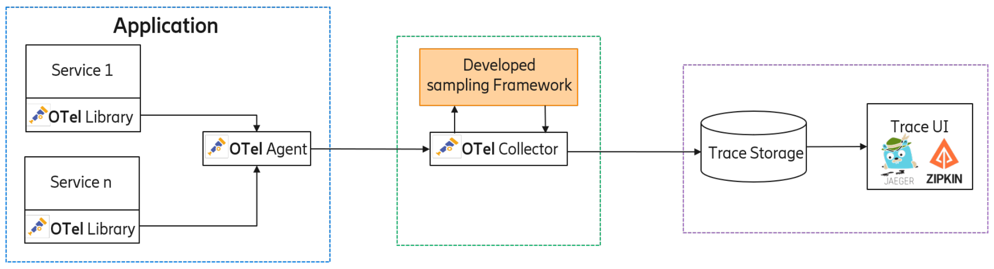


Figure 3.2: Interaction of the developed sampling framework with a standard OpenTelemetry deployment

but beneficial because the batch processor performs the batching of spans in order to improve the compression of the data and to reduce the number of outgoing connections while sending the data to the *kafka* exporter. Then the *kafka* exporter sends the data to a Kafka topic named *input_queue* in OTLP protobuf format (*oltp_proto*).

Our tail-based sampling framework reads the protobuf serialized spans and starts processing them. Upon reaching a sampling decision, the framework writes the spans of the traces to be sampled to another Kafka topic named *output_queue* in *oltp_proto* format. At this point, the *kafka* receiver in the *traces/output* pipeline reads the sampled spans from the *output_queue* topic and passes them to the batch processor. Finally, the batch processor forwards them to the *jaeger* exporter, which passes the sampled span to Jaeger, thus completing the process of tail-based sampling.

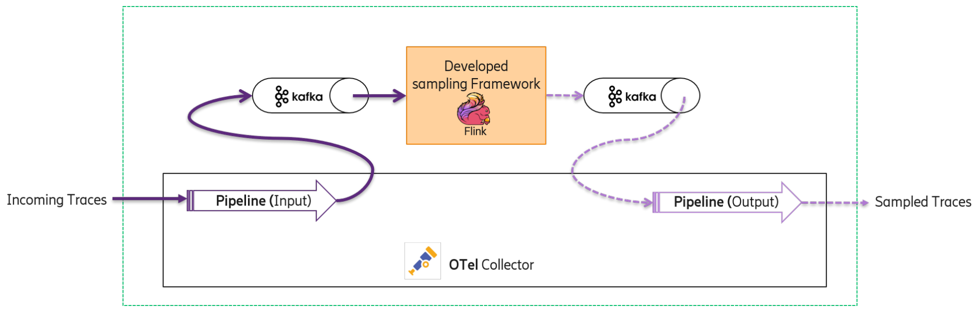


Figure 3.3: Dataflow between the developed sampling framework and Open-Telemetry collector

```
receivers:
    otlp:
        protocols:
            grpc:
    kafka:
        brokers:
          - kafka:9092
        topic: output_queue
        encoding: otlp_proto

exporters:
    jaeger:
        endpoint: jaeger-collector:14250
    kafka:
        brokers:
          - kafka:9092
        topic: input_queue
        encoding: otlp_proto

processors:
    batch:


service:
    pipelines:
        traces/input:
            receivers: [otlp]
            processors: [batch]
            exporters: [kafka]
        traces/output:
            receivers: [kafka]
            processors: [batch]
            exporters: [jaeger]
```

Code snippet 1: An OpenTelemetry collector pipeline configuration for using our tail-based sampling framework

3.4 Algorithm

As mentioned in Section 3.2.2, the tail-based sampling framework developed for this thesis is built using Apache Flink and runs on top of a Flink cluster. Upon submitting the tail-based sampling job to the Flink cluster, it reads two configuration files for initializing the framework with user-defined settings. The first configuration file contains information such as sampling budget, window size, and names of the input and output queues. The other configuration file accepts feature filter object definitions for performing feature selection of traces.

After the initialization phase, the framework starts processing the inbound stream of spans. It reads the incoming spans serialized in OTLP protobuf format from the input queue and creates a new data stream of spans after deserializing them into plain old Java objects (POJOs). The spans in this stream are keyed by *TraceId* which allows the aggregation of all spans of a particular trace into a single trace object. The framework handles out-of-order spans and maintains the causality relations of the spans of these trace objects. A new data stream is formed by using these newly created trace objects, which are partitioned using fixed-length windows for further processing.

Each of the traces in a particular window has to pass through a two-step evaluation process. In the first step, the framework checks for feature matches according to the feature filter object definitions. If a feature match is found, the trace is selected for sampling. In case there are no feature matches, the trace is evaluated in the next step, where the framework decides either to sample the trace or to add the trace to a candidate list for sampling.

The framework determines the outcome of the possibilities mentioned above by following a principle which we refer to as the Least Appearances First (LAF) principle. According to this principle, our framework prioritizes the traces that have appeared at the framework the least times. The philosophy behind the LAF principle is that the traces which appear more frequently represent the common execution paths of the system. Therefore, in order to capture unusual behaviors of the system, such as a newly introduced bug, the LAF principle prioritizes the traces which appear less frequently. As a result, a trace representing an erroneous transaction due to a newly introduced bug has more sampling priority than a trace representing a typical execution. The LAF principle requires the framework to track the frequency statistics of previously appeared traces, which is accomplished by using the state backends provided by Flink.

Once all of the traces of a window traverse this two-step evaluation process, the framework populates a list of traces to be sampled following the LAF principle known as candidate list. Finally, the traces are sampled from the list of candidates within the remaining sampling budget. Figure 3.4 shows a flowchart of the tail-based sampling process described above. Before sending the sampled traces back to the collector, the framework serializes them in OTLP protobuf format and writes them to the output queue.

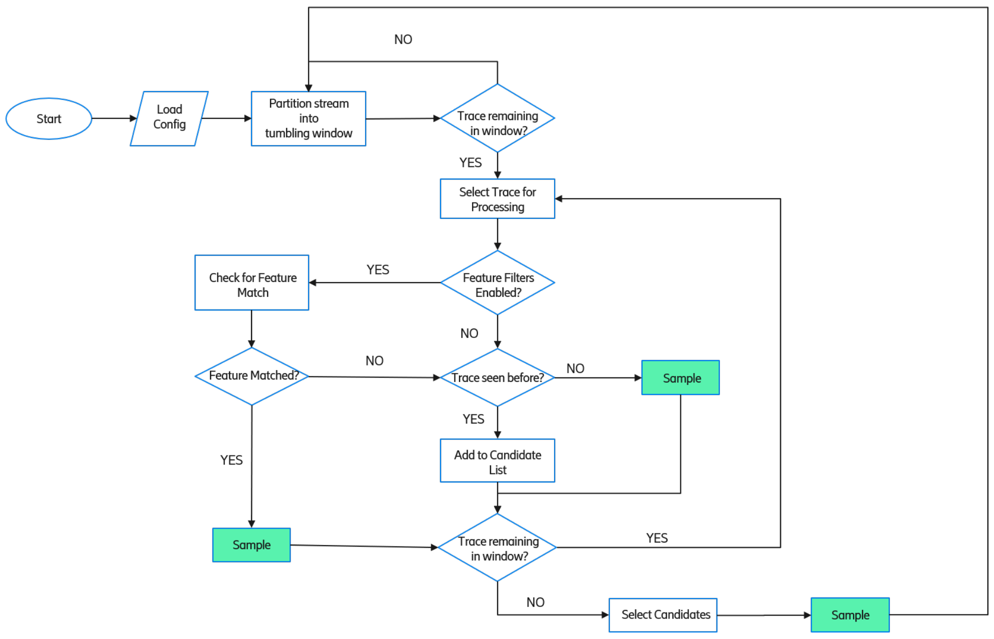


Figure 3.4: Flowchart of the developed tail-based sampling algorithm

3.5 Feature Selection

As described in Section 3.4, the developed tail-based sampling framework supports feature selection to sample traces. The classes performing feature selection are called feature filters, and the framework accepts zero or more feature filters provided in a configuration file. The framework allows simple configuration through a YAML file, which contains the object definitions of feature filters used in the sampling process. The framework ships with three predefined feature filters, namely *LatencyFilter*, *ResourceFilter*, and *ErrorCodeFilter*, and it is straightforward to add more custom feature filters.

As described in Section 3.4, the developed tail-based sampling framework supports feature selection to sample traces. The classes performing feature selection are called feature filters, and the framework accepts zero or more

feature filters provided in a configuration file. The framework allows simple configuration through a YAML file, which contains the object definitions of feature filters used in the sampling process. The framework ships with three predefined feature filters, namely *LatencyFilter*, *ResourceFilter*, and *ErrorCodeFilter*, and it is straightforward to add more custom feature filters.

Code snippet 2 shows an example configuration having three feature filters. This configuration sets up the framework to sample all the traces having latency beyond 500 milliseconds or the traces which were generated in *cart_service* from region *us_west_1*. To add a custom filter, the developer has to write his own implementation of a filter class extending a class named *FilterConfig*, which is the superclass of all feature filters. Finally, to use the newly defined feature filter, it is sufficient to add an entry to the configuration file, as Code snippet 2 demonstrates.

```yaml
LatencyFilter:
    args:
        duration: 500

ResourceFilter:
    args:
        service.name: cart_service
        region: us_west_1

CustomFilter:
    args:
        key: value
```

Code snippet 2: Configuration file for enabling feature filters

Chapter 4

Evaluation

In this chapter, we evaluate the overhead of distributed tracing on ESM and compare our tail-based sampling framework with the tail-based sampling processor of the OpenTelemetry collector. We measure the memory requirements and present a detailed comparison of available features of both tail-based sampling implementations.

4.1 Overhead of Tracing on ESM

In order to measure the performance overhead of distributed tracing on ESM, we instrumented a scheduler service in ESM that is used to schedule periodic jobs. We selected five different tasks that the scheduler service executes periodically at regular intervals. We measured the end-to-end latency of these tasks both before and after the introduction of distributed tracing. Figure 4.1 illustrates the findings from this experiment in a bar chart where the x-axis lists five different tasks, and the y-axis represents the end-to-end latency of the tasks in milliseconds. The green bars in the chart represent the measurements taken before enabling distributed tracing, while the red bars denote the measurements after enabling distributed tracing. The black lines on each bar represent the standard error of the corresponding latency measurements. In this figure, the red bars exhibit higher latencies than the corresponding green bars for all five tasks. The figure implies that the introduction of distributed tracing induces performance overhead on the instrumented application.

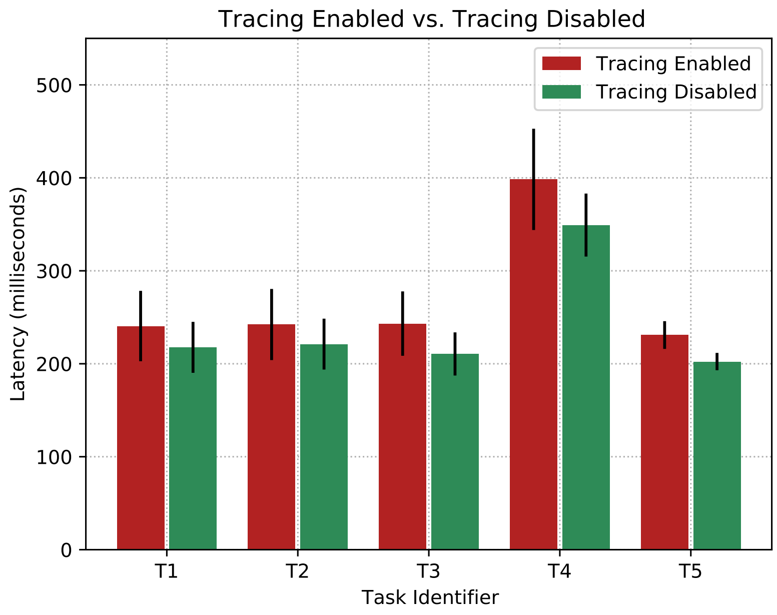


Figure 4.1: Latency comparison of the scheduled tasks with tracing toggled

Figure 4.2 shows the amount of overhead experienced by each of the instrumented tasks in terms of increase in latency. In this figure, the x-axis lists all the tasks, and the y-axis denotes the latency increase in percentage. According to this figure, the latency overhead of the scheduled tasks varies from 9% up to almost 16%.

4.2 Memory Requirements

In this section, we measure the memory requirements of the OpenTelemetry collector for performing tail-based sampling. For this evaluation, we consider three scenarios as follows: disabling sampling, performing tail-based sampling with the developed framework, and performing tail-based sampling using the tail-based sampling processor provided by OpenTelemetry. The measurements were conducted using the configurations listed in Table 4.1. In this table, spans/second refers to the number of spans sent to the OpenTelemetry collector instance per second. Window size or decision time is the waiting period before generating sampling output. Each measurement run lasted for 15 minutes, and the collector instance was allocated a dedicated CPU core running at 2 GHz to ensure stable operation of OpenTelemetry collector across different

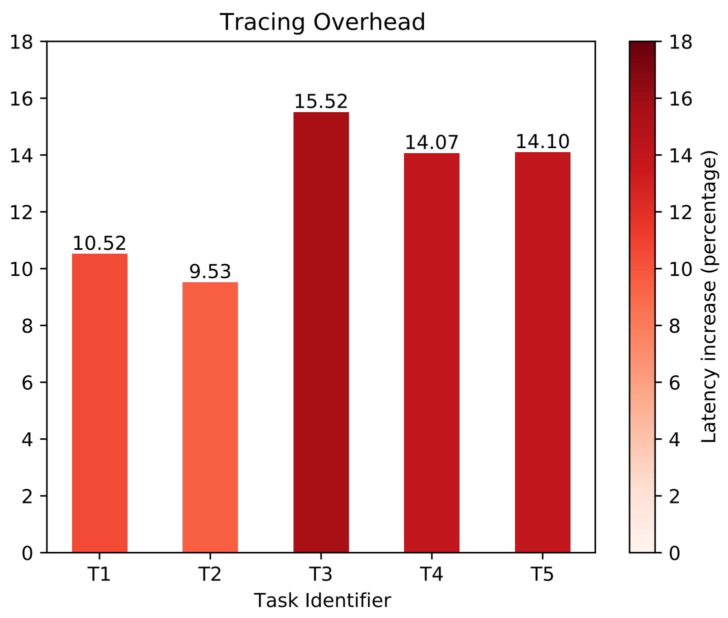


Figure 4.2: Latency increase of the scheduled tasks in percentage

measurement runs.

Criterion	**Value**
Spans/second	255
Window size/Decision time	1 second
Duration	15 minutes
CPU	1 core @ 2GHz

Table 4.1: Test configuration

4.2.1 Memory Consumption of OpenTelemetry Collector

Figure 4.3 compares the memory consumption of an OpenTelemetry collector instance in three different scenarios. In this figure, the x-axis represents the memory consumption of the OpenTelemetry collector instance in megabytes (MB), and the y-axis denotes the three different sampling scenarios described in Section 4.2. The topmost bar shows the memory consumption of the OpenTelemetry collector instance with sampling disabled, which is slightly more than

70 MB. The middle bar shows that while using the developed tail-based sampling framework, the memory consumption goes up by only a few megabytes but remains below 80 MB. Finally, the bottom bar shows that the memory consumption of the collector spikes up to almost 120 MB when using the OpenTelemetry tail-based sampling processor.

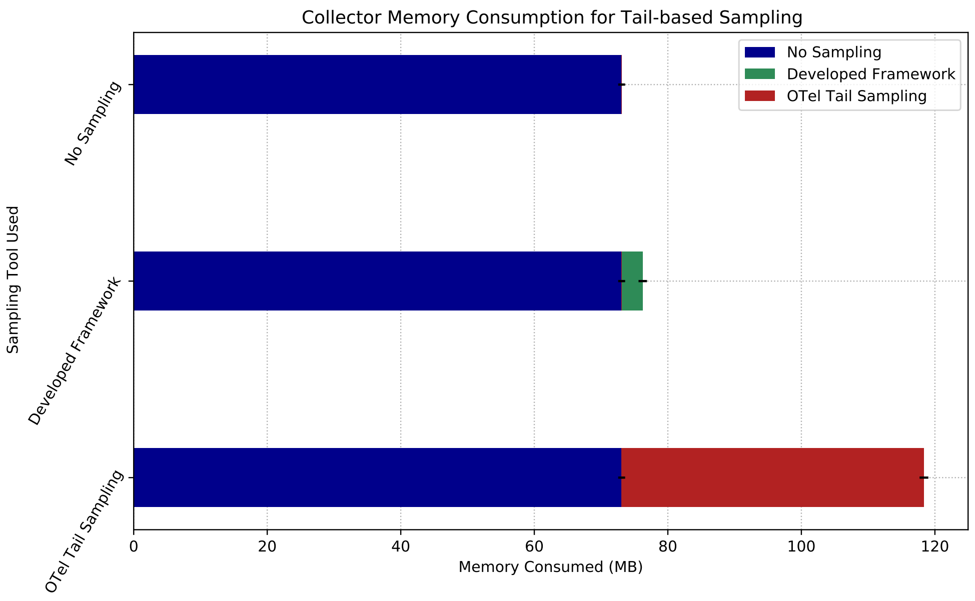


Figure 4.3: Comparison of memory consumption of an OpenTelemetry collector instance in different scenarios: without tail-based sampling, while using the developed tail-based sampling framework, and while using OpenTelemetry tail-based sampling processor

4.2.2 Memory Consumption of Flink Cluster

The developed tail-based sampling framework runs on top of a Flink cluster that has its own memory requirements. Figure 4.4 shows the memory consumed by the developed sampling framework where the x-axis represents time, and the y-axis denotes memory consumption. According to this figure, the developed framework consumes maximally 220 MB memory in the Flink cluster. The spikes in this figure are due to the garbage collection mechanism in Java. This result implies that despite consuming less memory than the OpenTelemetry tail-based sampling processor in an OpenTelemetry collector instance, according to Figure 4.3, our framework requires more memory to operate due to the additional memory requirement of the Flink cluster.

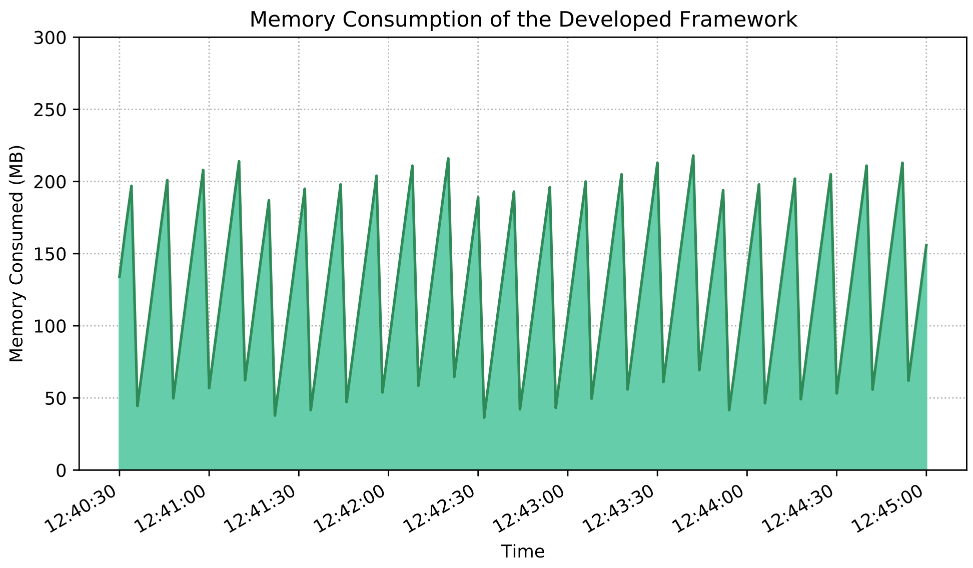


Figure 4.4: Memory consumption of developed tail-based sampling framework running on a Flink cluster

4.3 Comparison

The following table lists a detailed comparison of the features available in both tail-based sampling implementations.

Criterion	**OTel Implementation**	**Developed Framework**
Minimum decision latency	Seconds	Milliseconds
Supports collector scaling	No	Yes
Rate limiting	Spans/sec	Traces/sec
Rate limiting enforced	No	Yes

Table 4.2: Comparison between tail-based sampling implementation of Open-Telemetry and the developed tail-based sampling framework

Chapter 5

Discussion and Future Work

In this chapter, we examine and explore our research findings from the previous sections to answer the research questions listed in Section 1.3. Additionally, we list potential future directions for improving our tail-based sampling framework.

5.1 Overhead of Tracing on ESM

In our quest of investigating the overhead of distributed tracing on ESM, we discovered a significant increase in the end-to-end latencies of the tasks instrumented with distributed tracing. Figure 4.2 reported an average of around 12.75% latency increase across all five tasks. Head-based sampling can be employed in order to tackle this deterioration of performance. However, to ensure representative sampling of the traces during head-based sampling, it is necessary to define varying sampling probabilities for different endpoints of each service.

5.2 Capturing Insightful Traces

One of the main motivations behind performing tail-based sampling is to aid debugging and performance evaluation by storing insightful traces. Frequently appearing traces represent common cases of execution of a system. In contrast, the traces that appear rarely represent unusual behaviors of the system, and thus they are more beneficial for future analysis. Our tail-based sampling framework excels in this area due to its ability to collect rare traces. Currently, the LAF principle prioritizes the traces in ascending order of their occurrence. This policy ensures the sampling of uncommon traces, which is the distinguishing feature of our developed tail-based sampling framework compared to the tail-based sampling processor of OpenTelemetry.

5.3 Benefits over OpenTelemetry Tail-based Sampling Processor

As demonstrated in Section 4.2.1, our framework has a smaller memory footprint in the OpenTelemetry collector instance compared to the OpenTelemetry tail-based sampling processor. However, considering the additional memory requirement of the Flink cluster discussed in Section 4.2.2, our framework requires more memory to operate. Nevertheless, the developed framework offers the advantages listed in Table 4.2, which make it more effective than the tail-based sampling processor of OpenTelemetry, despite its excess memory requirement. The advantages of using our tail-based sampling framework are as follows.

Firstly, the minimum decision latency in our framework can be defined in milliseconds compared to seconds in the OpenTelemetry tail-based sampling processor. As a result, our framework can produce sampling output faster than the OpenTelemetry tail-based sampling processor.

Secondly, our tail-based sampling framework provides better performance at scale as it supports the scaling of the collectors. The developed framework does not rely on the collector instances to aggregate all spans of a single trace. Instead, it performs the aggregation of spans internally. Conversely, the tail-based sampling processor of OpenTelemetry expects all spans of a single trace to arrive at the same collector instance within a fixed time frame which complicates the scaling of the collectors.

The next advantage of using the developed tail-based sampling framework is related to the ease of configuration of rate limiting. Our framework enables rate limiting by specifying the number of traces to be sampled per second. In contrast, the OpenTelemetry tail-based sampling processor supports rate limiting by specifying the number of spans to be sampled per second. As a result, our framework relieves the developer from the responsibility of determining a suitable value to ensure sampling of all spans of a single trace.

Finally, rate limiting is strictly enforced by our tail-based sampling framework, which is one of the primary goals of performing any variety of sampling in the first place. However, the OpenTelemetry tail-based sampling processor does not strictly enforce rate limiting. Spans matching with any feature filter get sampled, regardless of the threshold defined in the rate limiting filter.

5.4 Compatibility

The framework developed for this thesis is compliant with the OpenTelemetry specifications. The internal representations of spans are constructed from the OTLP protobuf schemas. As a result, the developers can easily add custom filters for feature selection without learning new schemas. Moreover, the framework accepts and emits the spans in OTLP protobuf format, making the framework compatible with any existing OpenTelemetry deployments.

5.5 Future Work

One general improvement of the developed tail-based sampling framework would be optimizing the memory usage of the framework. This optimization can be achieved by using a compact representation of spans for state management of the stream processing. Furthermore, our implementation of a tail-based sampling framework prioritizes the traces based on their frequency following the LAF principle, as explained in Section 3.4. A future direction of work in our framework could be supporting custom algorithms in addition to the LAF principle for prioritizing traces. Another future work item could be extending the developed tail-based sampling framework to perform real-time data analytics on the incoming traces leveraging the underlying stream processing engine. Supporting real-time data analytics would allow our framework to generate custom metrics beneficial for the business context.

Chapter 6

Conclusion

In recent years, the growing popularity of microservice architecture has increased the importance of employing distributed tracing in order to enhance the observability of microservices-based applications. Nevertheless, establishing a balance between obtaining better visibility into the systems and reducing the overhead of distributed tracing has always been a barrier to the widespread adoption of distributed tracing in organizations. Moreover, the extra storage cost of persisting traces causes additional complications. These issues can be addressed by using various sampling techniques to reduce the number of traces generated and stored.

In this thesis, we used OpenTelemetry to instrument a microservices-based application for examining the overhead of introducing distributed tracing. Later, we developed a tail-based sampling framework that can perform tail-based sampling in any OpenTelemetry deployments.

Specifically, in the first part of this thesis, we surveyed the core concepts of distributed tracing and explained various sampling methods used in distributed tracing. Later, we introduced OpenTelemetry, an open-source vendor-neutral observability project aimed at unifying the end-to-end processing of telemetry data, including metrics, logs, and traces. We examined the formation of traces in OpenTelemetry and discussed the architecture of the OpenTelemetry collector, which is at the core of performing tail-based sampling.

Next, we examined the impact of introducing distributed tracing to a microservices-based application. Our experiments demonstrated latency overheads ranging from 9% to 16% after the introduction of distributed tracing. To tackle this deterioration of performance, we recommended the usage of head-based sampling.

Finally, we explored the prospective of using stream processing to perform tail-based sampling. We elected stream processing to perform tail-based sampling as stream processing engines can efficiently handle a high volume of streaming data in real-time, which is essential for coping with traces generated by services serving a high traffic volume. We used an industry-leading open-source stream processing framework named Apache Flink to develop our tail-based sampling framework, and the built framework is compliant with OpenTelemetry specifications. As a result, the framework is readily compatible with any existing OpenTelemetry deployments.

Our experiments have demonstrated promising performance of the developed framework by saving approximately a third of memory-based storage compared to an OpenTelemetry tail-based sampling module. Furthermore, our framework provides additional features that are currently missing in the OpenTelemetry tail-based sampling module.

Bibliography

[1] C. Richardson, *Microservices Patterns: With examples in Java*. Manning Publications, 2018, ISBN: 9781617294549. [Online]. Available: https://books.google.fi/books?id=UeK1swEACAAJ.

[2] C. Richardson, *What are microservices?*, https://microservices.io/, Online; Accessed April 14, 2021.

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

[4] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, “Managing security of virtual machine images in a cloud environment”, in *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009, pp. 91–96.

[5] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes”, *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014. DOI: 10.1109/MCC.2014.51.

[6] A. Khan, “Key characteristics of a container orchestration platform to enable a modern application”, *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017. DOI: 10.1109/MCC.2017.4250933.

[7] M. Pearce, S. Zeadally, and R. Hunt, “Virtualization: Issues, security threats, and solutions”, *ACM Comput. Surv.*, vol. 45, no. 2, Mar. 2013, ISSN: 0360-0300. DOI: 10.1145/2431211.2431216. [Online]. Available: https://doi.org/10.1145/2431211.2431216.

[8] R. Bankston and J. Guo, “Performance of container network technologies in cloud environments”, in *2018 IEEE International Conference on Electro/Information Technology (EIT)*, 2018, pp. 0277–0283. DOI: 10.1109/EIT.2018.8500285.

[9] D. Jaramillo, D. V. Nguyen, and R. Smart, “Leveraging microservices architecture by using docker technology”, in *SoutheastCon 2016*, 2016, pp. 1–5. DOI: 10.1109/SECON.2016.7506647.

[10] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments”, in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013, pp. 233–240. DOI: 10.1109/PDP.2013.41.

[11] “Ieee standard glossary of software engineering terminology”, *IEEE Std 610.12-1990*, pp. 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064.

[12] C. Sridharan, *Distributed Systems Observability: A Guide to Building Robust Systems*. O’Reilly Media, 2018, ISBN: 9781492033424. [Online]. Available: https://books.google.fi/books?id=07EswAEACAAJ.

[13] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, “Performance debugging for distributed systems of black boxes”, *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 74–89, 2003.

[14] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, “Wap5: Black-box performance debugging for wide-area systems”, in *Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 347–356.

[15] J. Turnbull, *The Art of Monitoring*. James Turnbull, 2014, ISBN: 9780988820241. [Online]. Available: https://books.google.fi/books?id=w5QfDAAAQBAJ.

[16] M. Gopal, *Modern Control System Theory*. USA: Halsted Press, 1984, ISBN: 0470274247.

[17] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic instrumentation of production systems”, in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’04, Boston, MA: USENIX Association, 2004, p. 2.

[18] B. H. Sigelman, L. A. Barroso, M. Burrows, *et al.*, “Dapper, a large-scale distributed systems tracing infrastructure”, 2010. [Online]. Available: https://research.google/pubs/pub36356/.

[19] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, “Magpie: Online modelling and performance-aware systems”, in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS’03, Lihue, Hawaii: USENIX Association, 2003, p. 15.

[20] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, “X-trace: A pervasive network tracing framework”, in *Proceedings of the 4th USENIX Conference on Networked Systems Design &amp; Implementation*, ser. NSDI’07, Cambridge, MA: USENIX Association, 2007, p. 20.

[21] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, “The mystery machine: End-to-end performance analysis of large-scale internet services”, in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’14, Broomfield, CO: USENIX Association, 2014, pp. 217–231, ISBN: 9781931971164.

[22] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, “Using magpie for request extraction and workload modelling”, in *Proceedings of the 6th Conference on Symposium on Operating Systems Design &amp; Implementation - Volume 6*, ser. OSDI’04, San Francisco, CA: USENIX Association, 2004, p. 18.

[23] S. Kanzhelev, M. McLean, A. Reitbauer, B. Drutu, N. Molnar, and Y. Shkuro, “Trace context”, W3C, W3C Working Draft, Feb. 2020, https://www.w3.org/TR/trace-context/.

[24] L. Lamport, “Time, clocks, and the ordering of events in a distributed system”, *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978, ISSN: 0001-0782. DOI: 10.1145/359545.359563. [Online]. Available: https://doi.org/10.1145/359545.359563.

[25] T. Holl, P. Klocke, F. Franzen, and J. Kirsch, “Kernel-assisted debugging of linux applications”, in *Proceedings of the 2nd Reversing and Offensive-Oriented Trends Symposium*, ser. ROOTS ’18, Vienna, Austria: Association for Computing Machinery, 2018, ISBN: 9781450361712. DOI: 10.1145/3289595.3289596. [Online]. Available: https://doi.org/10.1145/3289595.3289596.

[26] J. Keniston, A. Mavinakayanahalli, and V. Prasad, “Ptrace , utrace , uprobes : Lightweight , dynamic tracing of user apps”, 2007. [Online]. Available: https://landley.net/kdocs/ols/2007/ols2007v1-pages-215-224.pdf.

[27] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, “Probing the guts of kprobes”, 2010. [Online]. Available: https://landley.net/kdocs/ols/2006/ols2006v2-pages-109-124.pdf.

[28] Z. Zhang, J. Zhan, Y. Li, L. Wang, D. Meng, and B. Sang, “Precise request tracing and performance debugging for multi-tier services of black boxes”, in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, 2009, pp. 337–346. DOI: 10.1109/DSN.2009.5270321.

[29] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O’Reilly Media, Inc., 2005, ISBN: 0596005903.

[30] B. Gregg and J. Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, 1st. USA: Prentice Hall Press, 2011, ISBN: 0132091518.

[31] D. Narayanan, “End-to-end tracing considered essential”, in *Proceedings of High Performance Transaction Systems–Eleventh Biennial Workshop (HPTS’05)*, 2005.

[32] B. Miller, “Dpm: A measurement system for distributed programs”, *Computers, IEEE Transactions on*, vol. 37, pp. 243–248, Mar. 1988. DOI: 10.1109/12.2157.

[33] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, “E2eprof: Automated end-to-end performance management for enterprise systems”, in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)*, 2007, pp. 749–758. DOI: 10.1109/DSN.2007.38.

[34] E. Koskinen and J. Jannotti, “Borderpatrol: Isolating events for black-box tracing”, in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys ’08, Glasgow, Scotland UK: Association for Computing Machinery, 2008, pp. 191–203, ISBN: 9781605580135. DOI: 10.1145/1352592.1352613. [Online]. Available: https://doi.org/10.1145/1352592.1352613.

[35] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, “Pinpoint: Problem determination in large, dynamic internet services”, in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 595–604. DOI: 10.1109/DSN.2002.1029005.

[36] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer, “Using runtime paths for macro analysis”, in *9th Workshop on Hot Topics in Operating Systems*, May 2003. [Online]. Available: https://www.microsoft.com/en-us/research/publication/using-runtime-paths-for-macro-analysis/.

[37] H. Zhang, J. Rhee, N. Arora, *et al.*, “Clue: System trace analytics for cloud service performance diagnosis”, in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–9. DOI: 10.1109/NOMS.2014.6838348.

[38] S. E. Perl and W. E. Weihl, “Performance assertion checking”, *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 134–145, Dec. 1993, ISSN: 0163-5980. DOI: 10.1145/173668.168630. [Online]. Available: https://doi.org/10.1145/173668.168630.

[39] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, “Pip: Detecting the unexpected in distributed systems”, in *Proceedings of the 3rd Conference on Networked Systems Design &amp; Implementation - Volume 3*, ser. NSDI’06, San Jose, CA: USENIX Association, 2006, p. 9.

[40] B. Miller, M. Callaghan, J. Cargille, *et al.*, “The paradyn parallel performance measurement tool”, *Computer*, vol. 28, no. 11, pp. 37–46, 1995. DOI: 10.1109/2.471178.

[41] T. Gschwind, K. Eshghi, P. Garg, and K. Wurster, “Webmon: A performance profiler for web transactions”, in *Proceedings Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2002)*, 2002, pp. 171–176. DOI: 10.1109/WECWIS.2002.1021256.

[42] E. Thereska, B. Salmon, J. Strunk, *et al.*, “Stardust: Tracking activity in a distributed storage system”, in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’06/Performance ’06, Saint Malo, France: Association for Computing Machinery, 2006, pp. 3–14, ISBN: 1595933190. DOI: 10.1145/1140277.1140280. [Online]. Available: https://doi.org/10.1145/1140277.1140280.

[43] J. Kaldor, J. Mace, M. Bejda, *et al.*, “Canopy: An end-to-end performance tracing and analysis system”, in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17, Shanghai, China: Association for Computing Machinery, 2017, pp. 34–50, ISBN:

9781450350853. DOI: 10.1145/3132747.3132749. [Online]. Available: https://doi.org/10.1145/3132747.3132749.

[44] Apache SkyWalking, *Apache SkyWalking*, https://skywalking.apache.org/, Online; Accessed April 14, 2021.

[45] Zipkin, *Zipkin*, https://zipkin.io/, Online; Accessed April 14, 2021.

[46] Jaeger, *Sampling*, https://www.jaegertracing.io/docs/1.23/sampling/, Online; Accessed April 14, 2021.

[47] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering", in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19, Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 312–324, ISBN: 9781450369732. DOI: 10.1145/3357223.3362736. [Online]. Available: https://doi.org/10.1145/3357223.3362736.

[48] P. Las-Casas, J. Mace, D. Guedes, and R. Fonseca, "Weighted sampling of execution traces: Capturing more needles and less hay", in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18, Carlsbad, CA, USA: Association for Computing Machinery, 2018, pp. 326–332, ISBN: 9781450360111. DOI: 10.1145/3267809.3267841. [Online]. Available: https://doi.org/10.1145/3267809.3267841.

[49] J. Mace and R. Fonseca, "Revisiting end-to-end trace comparison with graph kernels", 2013. [Online]. Available: https://cs.brown.edu/~jcmace/papers/mace13revisiting.pdf.

[50] Natasha Woods, *OpenTracing joins the Cloud Native Computing Foundation*, https://www.cncf.io/blog/2016/10/11/opentracing-joins-the-cloud-native-computing-foundation/, Online; Accessed April 14, 2021.

[51] Pritam Shah and Morgan McLean, *The value of OpenCensus*, https://opensource.googleblog.com/2018/03/the-value-of-opencensus.html, Online; Accessed April 14, 2021.

[52] Ben Sigelman, *A brief history of OpenTelemetry (So Far)*, https://www.cncf.io/blog/2019/05/21/a-brief-history-of-opentelemetry-so-far/, Online; Accessed April 14, 2021.

[53] OpenTelemetry, *OpenTelemetry: An observability framework for cloud-native software*, https://opentelemetry.io/, Online; Accessed April 14, 2021.

[54] ——, *OpenTelemetry Specification*, https://github.com/open-telemetry/opentelemetry-specification, Online; Accessed April 14, 2021.

[55] B. Burns, *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*, 1st. O'Reilly Media, Inc., 2018, ISBN: 1491983647.

[56] Spring, *Spring Cloud Sleuth*, https://spring.io/projects/spring-cloud-sleuth, Online; Accessed April 14, 2021.

[57] Flink, *Apache Flink™ — Stateful Computations over Data Streams*, https://flink.apache.org/, Online; Accessed April 14, 2021.

[58] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine", *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015. [Online]. Available: http://sites.computer.org/debull/A15dec/p28.pdf.

[59] T. Akidau, S. Chernyak, and R. Lax, *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*, 1st. O'Reilly Media, Inc., 2018, ISBN: 1492034142.

[60] T. Akidau, E. Begoli, S. Chernyak, *et al.*, "Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow", *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 3135–3147, Jul. 2021, ISSN: 2150-8097. DOI: 10.14778/3476311.3476389. [Online]. Available: https://doi.org/10.14778/3476311.3476389.

[61] J. Kreps, N. Narkhede, and J. Rao, "Kafka : A distributed messaging system for log processing", 2011. [Online]. Available: http://notes.stephenholiday.com/Kafka.pdf.

[62] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper", in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '17, Barcelona, Spain: Association for Computing Machinery, 2017, pp. 227–238, ISBN:

9781450350655. DOI: 10.1145/3093742.3093908. [Online]. Available: https://doi.org/10.1145/3093742.3093908.

TRITA-EECS-EX-2021:820