



Titre: Tracing Low Latency Microservices
Title:

Auteur: Eya-Tom Augustin Sangam
Author:

Date: 2023

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Sangam, E.-T. A. (2023). Tracing Low Latency Microservices [Mémoire de maîtrise,
Citation: Polytechnique Montréal]. PolyPublie. <https://publications.polymtl.ca/57053/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/57053/>
PolyPublie URL:

**Directeurs de
recherche:** Michel Dagenais
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Tracing low latency microservices

EYA-TOM AUGUSTIN SANGAM

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie informatique

Décembre 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Tracing low latency microservices

présenté par **Eya-Tom Augustin SANGAM**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Daniel ALOISE, président

Michel DAGENAIS, membre et directeur de recherche

Soumaya CHERKAOUI, membre

DEDICATION

To my family and all my friends . . .

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Michel Dagenais. His belief in my potential and his efforts in securing the necessary funds for my project have been invaluable to my research. His insightful feedbacks and patient mentoring have greatly enriched my academic experience and have shaped my growth both as a researcher and as a person.

I would also like to express my sincere appreciation to Hadi Dewan, a Software Architect at Ciena. His invaluable collaboration, provision of practical use cases for testing, and his insightful guidance have significantly contributed to the success of my research. His expertise and real-world insights have deepened my understanding of the field and have enriched my learning experience.

I extend my appreciation to all my colleagues at the DORSAL laboratory. Their camaraderie, support, and intellectual stimulation have made my time at the lab both productive and enjoyable.

I am profoundly grateful to Polytechnique Montreal and the Department of Computer and Software Engineering for the invaluable opportunity to study and grow within such a prestigious institution. The scholarship they awarded me has been fundamental to my journey.

I would like to extend my gratitude to Ericsson, Ciena, AMD, EfficiOS, NSERC, and Prompt for their generous funding. Their financial support has been instrumental in carrying out this project and I am sincerely thankful for their contribution.

Lastly, I am forever indebted to my parents and family for their endless love, support, and encouragement throughout my studies. Their unwavering belief in my abilities and their sacrifices to provide me with the opportunity to pursue my education have been the cornerstone of all my achievements.

RÉSUMÉ

Les architectures en microservices décomposent les applications en petits services autonomes. Ils permettent de diminuer la complexité des systèmes basés sur des architecture monolithiques en offrant des mises à jour simplifiées, un haut degré de flexibilité et plus de robustesse. Cependant, leur distribution à travers les systèmes complique le débogage et la surveillance, rendant le traçage et la surveillance de la santé vitaux pour assurer des performances optimales. Il devient crucial pour les administrateurs système de recueillir des données de télémétrie (DT), provenant des microservices, pour garantir une surveillance, optimisation et fiabilité du système. En procédant ainsi, il devient facile d'identifier les goulots d'étranglement de performance, détecter les anomalies et prendre des décisions basées sur les données.

Ce projet de recherche se concentre sur deux objectifs cruciaux. Le premier objectif est de collecter des DT à partir de microservices qui nécessitent une faible latence pour leurs opérations. Ces microservices sont généralement équipés de ressources minimales telles que le stockage et la mémoire vive. Deuxièmement, le projet vise à suggérer des techniques efficaces pour analyser les DT recueillies sur plusieurs machines, avec l'objectif ultime de détecter les goulots d'étranglement de performance.

Le défi ici réside dans la gestion efficace de ces ressources limitées, tout en garantissant que les données nécessaires à la surveillance et à l'optimisation sont collectées et traitées de manière efficace. La clé pour y parvenir est de mettre en œuvre des stratégies de filtrage qui réduisent la quantité de TD sauvegardée, réduisent la quantité de données envoyées sur le réseau, et minimisent l'impact sur les performances du système pendant la collecte de données.

Pour répondre à ces exigences, nous suggérons la combinaison de deux technologies, OpenTelemetry et LTTng. Le processus implique deux phases distinctes: une phase en ligne et une phase hors ligne. La phase en ligne implique la structuration et la collecte de DT avec OpenTelemetry, qui sont ensuite stockées dans des fichiers Common Trace Format (CTF) avec LTTng. Cette phase garantit que la collecte de données est effectuée de manière structurée, facilitant ainsi le processus d'analyse ultérieur. La phase hors ligne, en revanche, se concentre sur la manière dont les analyses des données collectées sont effectuées. Cette phase permet l'identification des anomalies du système et des goulots d'étranglement, et soutient la prise de décision basée sur les données.

ABSTRACT

Microservices architectures, which break applications into small autonomous services, address the complexities of monolithic architectures and offers simplified updates, a high degree of flexibility and robustness. Low latency microservices, designed for minimal delay, are essential in real-time response industries like high-frequency trading and telecommunications. However, their distribution across several networked systems complicates debugging and monitoring, making tracing and health monitoring vital for optimal performance. It becomes crucial for system administrators to gather telemetry data (TD), from microservices, to ensure seamless monitoring, optimization, and reliability of the system. By doing so, they can easily pinpoint performance bottlenecks, detect anomalies, and make data-driven decisions.

This study aims to achieve two important objectives. The first goal is to gather TD from microservices that necessitate low latency for their operations. These microservices are generally provided with minimal resources, such as limited storage and memory. Secondly, the project aims to suggest efficient techniques for analyzing the gathered TD across multiple hosts, with the ultimate objective of detecting performance bottlenecks.

The challenge here lies in effectively managing these limited resources, while ensuring the data required for monitoring and optimization is collected and processed in an efficient manner. The key to achieving this is to implement filtering strategies that reduce the amount of TD saved, reduce the amount of data sent across the network, and minimize the impact on system performance during data collection.

To efficiently collect and analyze TD from microservices, we recommend using OpenTelemetry and LTTng technologies. The process involves two distinct phases: an online phase and an offline phase. The online phase involves the structuring and collection of TD with OpenTelemetry, which is then stored in Common Trace Format (CTF) files with LTTng. This phase ensures that the collection of data is achieved in a structured manner, thereby facilitating the analysis process later on. The offline phase, on the other hand, focuses on how the analyses of the collected data are performed. This phase enables the identification of system anomalies, and bottlenecks, and supports data-driven decision-making.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vi
TABLE OF CONTENTS	vii
LIST OF TABLES	ix
LIST OF FIGURES	x
LIST OF SYMBOLS AND ACRONYMS	xi
CHAPTER 1 INTRODUCTION	1
1.1 Definitions and basic concepts	1
1.1.1 Tracing	1
1.1.2 Telemetry data	2
1.1.3 Observability backends	6
1.2 Elements of the problem	6
1.3 Research objectives	8
1.4 Outline of the research	8
CHAPTER 2 LITERATURE REVIEW	9
2.1 Distributed tracers	9
2.2 Kernel and userspace tracers	19
2.3 Observability agents	26
2.4 Telemetry data analysis	27
2.4.1 Observability backends	27
2.4.2 Other tools and techniques	32
2.5 Summary	34
CHAPTER 3 ENVIRONMENT SETUP	36
3.1 Hardware and software configuration	36

3.2	Developed tools	36
CHAPTER 4 ARTICLE 1: TRACING LOW LATENCY MICROSERVICES		38
4.1	INTRODUCTION	39
4.2	DEFINITIONS AND NOMENCLATURE	41
4.2.1	Tracing	41
4.2.2	Telemetry data	42
4.2.3	Observability backends	46
4.3	RELATED WORK	46
4.3.1	Distributed tracing	46
4.3.2	Low latency software tracing	49
4.3.3	Other tools and techniques	50
4.3.4	Summary	52
4.4	COMBINING OPENTELEMETRY AND LTTNG	54
4.4.1	Using OTel instrumentation	54
4.4.2	Using custom instrumentation	58
4.5	EVALUATION	60
4.5.1	Overhead of collecting request traces	61
4.5.2	Overhead of collecting application metrics	66
4.5.3	Overhead of collecting application logs	67
4.5.4	Overhead of collecting host metrics	67
4.5.5	Discussion and proposed tracing architecture	67
4.6	USE CASE: TRACING ZEROMQ BASED MICROSERVICES	69
4.7	CONCLUSIONS AND FUTURE WORK	72
4.8	ACKNOWLEDGMENTS	74
CHAPTER 5 GENERAL DISCUSSION		75
5.1	Review of the thesis objectives	75
5.2	Comparison with other approaches	75
CHAPTER 6 CONCLUSION		77
6.1	Summary of Works	77
6.2	Limitations	77
6.3	Future Research	78
REFERENCES		79

LIST OF TABLES

Table 4.1	Benchmark configuration options	62
Table 4.2	Requests trace collection benchmarks	63
Table 4.4	Flooding OTel ring buffers	66
Table 4.5	Application metrics collection benchmarks	68
Table 4.6	Application logs collection benchmarks	68
Table B.1	OpenTelemetry semantic conventions for servers general attributes . .	98

LIST OF FIGURES

Figure 2.1	Jaeger deployments options	14
Figure 2.2	OpenTelemetry collector features	19
Figure 2.3	Components of LTTng	23
Figure 2.4	Jaeger UI showing a sample distributed trace	29
Figure 4.1	Spans tree and swimlane visualizations	43
Figure 4.2	Trace Compass Control Flow View	53
Figure 4.3	Different ways of combining OTel and LTTng to collect application telemetry data	55
Figure 4.4	Tracing architecture	70
Figure 4.5	ZeroMQ client proxy server demo application	71
Figure 4.6	OTel Replayer architecture	73

LIST OF SYMBOLS AND ACRONYMS

API	Application Programming Interface
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
CTF	Common Trace Format
eBPF	extended Berkeley Packet Filter
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
LTNg	Linux Trace Toolkit: next generation
OB	Observability backend
OTel	OpenTelemetry
OTLP	OpenTelemetry Protocol
SDK	Software Development Kit
TD	Telemetry Data

CHAPTER 1 INTRODUCTION

Microservices, often referred to as the microservices architecture, represent an architectural approach that breaks down an application into a suite of small, autonomous services, each centered around specific business capabilities. These services, each deployable, operational, and scalable on their own, offer a high degree of flexibility and robustness. The shift towards microservices is driven by the challenges posed by monolithic architectures, which become increasingly complex to comprehend, develop, test, and deploy as applications expand. Microservices address these issues with their modular and independent nature, simplifying updates and mitigating risks.

Low latency microservices refer to services designed and optimized to respond to requests or process data with minimal delay. This is especially important in industries and applications where real-time response or processing is required. Examples include financial services for high-frequency trading, telecommunications, ad-tech, and real-time gaming.

In a typical microservices-based architecture, a single business process may involve numerous microservices, possibly running on distinct systems or containers. This dispersion adds a layer of complexity to debugging and monitoring, compared to their monolithic counterparts. Thus, tracing and monitoring has emerged as crucial elements in maintaining optimal performance and functionality in contemporary microservice architectures. This involves monitoring the health, performance, and interactions of the various software and hardware components that constitute the entire system.

The objective of this thesis is to propose a collection of tools and methodologies for gathering telemetry data (TD), from low latency microservices, and subsequently analyzing them. Such TD are vital for effective microservices monitoring.

1.1 Definitions and basic concepts

1.1.1 Tracing

Tracing is a technique used in software debugging and performance analysis to record the sequence of operations or function calls made by a program or a system. It allows developers and system administrators to understand the behavior of a system during its execution [1].

Tracers are the tools or utilities that facilitate this process. They monitor and record the events of interest in a system. These events can range from system calls, function entry/exit,

interrupts, and exceptions, to other specific application or system-level events.

The study will concentrate on three distinct categories of tracers:

Kernel Tracers These tracers monitor activities at the kernel level. They can provide valuable insights into how the operating system is interacting with the hardware and managing resources. Examples of Kernel tracers include **Ftrace** and **LTTng**.

Userspace Tracers These tracers operate at the application level, tracking the behavior of user programs. They can help developers understand the execution flow of their code, identify performance bottlenecks, or debug problems. Examples include **ltrace** and **LTTng-UST**.

Distributed Tracers These are used in distributed systems or microservices architectures. They help in understanding the interaction between different services in a system, which can be crucial for debugging complex distributed issues. Examples include **OpenTelemetry** clients.

Userspace and kernel tracers frequently depend on tracepoints for their operations. A **tracepoint** is a statement placed directly in the source code of the kernel or the application that provides a hook to invoke a probe. A **probe** is a function that is called whenever the tracepoint is encountered at runtime. A probe usually performs a custom task and is either implemented by the tracer or by the user. Typically, a probe is small and fast, which keeps it from slowing down the system. An **event** marks the encounter of a tracepoint at runtime. Depending on the kind of tracing, the event can have a logical meaning, such as a context switch, or can simply represent a location in the code, such as a function entry or exit. An event is instantaneous, without any duration and is typically marked with a timestamp for accurate tracking. Traces are a record of a series of events that have happened in the system.

1.1.2 Telemetry data

While the term “telemetry” has been in use for decades within various fields, like space exploration, the **OpenTelemetry** project certainly has played a significant role in popularizing it in the context of software observability and distributed systems. In this document, telemetry data (TD) refers to request traces, host and application metrics, and logs.

Request traces

Request traces, also known as distributed traces, are a crucial component for understanding the entire process when a request is made to an application. This holds true whether your

application is a single-database monolith or a complex network of services. By providing a comprehensive view of the complete “path” that a request follows in your application, traces enable you to get a clear understanding of the process [2].

Spans serve as the foundational building blocks of request traces. Essentially, a span encapsulates a single operation or unit of work performed within the system. The encapsulation may include, but is not limited to, the following components:

- A specific name identifying the span.
- Parent span ID, which remains empty for root spans, indicating the hierarchical relationship.
- Start and end timestamps, providing the exact time markers for the span’s life cycle.
- A context, which holds the trace identifiers and options that are propagated to child spans.
- Attributes, which are key-value pairs that provide additional details regarding the span.
- Events, which are time-stamped annotations on the span to describe noteworthy moments in the span’s lifetime.
- Links, which provide a connection between the span and other spans, which may be in the same or different traces.
- Status, an indicator of the outcome of the span - success, error, or cancellation.

Figure 4.1 provides an illustrative example of traces and spans.

Request tracing and context propagation Context propagation in request tracing refers to the process of passing along metadata from one service to another in a distributed system. This metadata, often referred to as the “context”, typically includes trace identifiers, identifiers of the individual operations, and other relevant information.

Context Propagation is the core concept that enables distributed tracing. The purpose of context propagation is to create complete, end-to-end traces of transactions as they traverse through various microservices in a distributed system. This allows for easier debugging, improved understanding of system behavior, and better performance analysis [3].

In a practical sense, context propagation usually involves adding specific headers to HTTP requests or messages shared between services. These headers are then read by

the receiving service and can be used to create a child span in its tracing system, thus creating a link between the services in the trace.

Host metrics

Host metrics allow you to monitor the performance and resource allocation of the hosts and processes that your critical applications depend on. These metrics provide a fundamental understanding of how your host resources are being utilized and can help identify potential performance issues or bottlenecks in your system. Host metrics typically focus on the fundamental resources of your system. Here are some common examples [4] :

- **CPU Usage:** Refers to the percentage of CPU resources in use and can be further broken down into user time, system time, and idle time.
- **Memory Usage:** The amount of memory being used versus the total available. This can include metrics for used memory, free memory, cached memory, and memory available for applications.
- **Disk I/O:** The amount of input and output operations happening on your disk. This can help identify if your disk has become a bottleneck.
- **Network Traffic:** Data such as the rate of data transmission and reception, packet loss, and network errors can help identify network-related issues.
- **Load Average:** A measure of the workload of the CPU and I/O system. It gives the average number of jobs in the execution queue over a certain period.
- **Disk Usage:** The amount of disk space used versus total available. This can help identify if you're running low on disk space.
- **Number of Running Processes:** The count of processes currently running on the host.
- **Number of Open Files:** The count of files currently being accessed.
- **Swap Usage:** The amount of swap space used versus total available.

Application metrics

An application metric is a real-time quantification of a service's performance [5]. The instant at which these quantifications are recorded is termed a metric event. This event comprises not only the measurement but also the timestamp of its capture and any pertinent metadata.

Metrics related to application performance and its request handling serve as vital barometers of system availability and efficiency. Other custom metrics can reveal the correlation between availability markers and user experience or business operations. The data collected from these measurements can be used to raise alerts during service disruptions or schedule automatic deployments in response to a rise in traffic.

Here are some examples [6]:

- Request Rate: The number of requests your application is handling per unit of time. High request rates could indicate a high load or potentially even a denial-of-service attack.
- Error Rate: The number of requests resulting in an error. This could indicate issues with your application code or downstream services.
- Response Time: The time it takes for your application to respond to a request. High response times could indicate performance issues.
- Throughput: The amount of data your application is processing in a given amount of time.
- Garbage Collection Metrics: If your application uses a garbage-collected language like Java or Go, metrics like the frequency and duration of garbage collections can be important indicators of performance.
- Database Query Times: The time taken to execute database queries. Long query times could indicate database performance issues or inefficient queries.
- Cache Hit/Miss Rates: If your application uses caching, the cache hit and miss rates can tell you how effectively your cache is being utilized.
- Queue Depth: If your application uses queues, the depth of the queue (number of messages waiting to be processed) can indicate if your application is able to keep up with the load.

Application logs

A log is a timestamped text record, either structured (recommended) or unstructured, with metadata. Most programming languages have built-in logging capabilities, or well-known widely used logging libraries. While logs can function as a standalone data source, they can also be appended to spans. Logs often contain detailed debugging/diagnostic info, such as

inputs to an operation, the result of the operation, and any supporting metadata for that operation [7].

1.1.3 Observability backends

Observability lets us understand a system from the outside, by letting us ask questions about that system without knowing its inner workings. Furthermore, it allows us to easily troubleshoot and handle novel problems (i.e. “unknown unknowns”), and helps us answer the question “Why is this happening”. Observability heavily relies on TD [8].

An Observability Backend (OB) is a platform that collects, stores, analyzes, and helps visualize TD. The main goal of an OB is to provide a comprehensive view of your system’s state and behavior over time. This helps in performance optimization, troubleshooting, and understanding the overall health of your system. OBs provide features (not necessarily all) for querying and visualizing TD, setting alerts based on specific conditions, and often include support for advanced analytics and machine learning, to detect anomalies and predict system behavior.

OBs can be categorized into three main types: metrics backends or time series databases, log backends or column stores, and request trace backends or request trace databases. Metric backends focus on storing and analyzing numerical data that measures system or application attributes over time, aiding in real-time monitoring, alerting, and performance trend analysis. Log backends handle the storage and analysis of log entries, providing detailed contextual information about events. Lastly, request trace backends focus on the storage and analysis of traces, which are records of a request path through the system.

Each type of OB is designed to handle a specific type of observability data. However, as shown in this recent study [9], many modern observability services combine all three types to provide a more comprehensive view of the system behavior.

1.2 Elements of the problem

Numerous tools are available to assist in the collection, visualization, and analysis of TD with the aim of identifying and rectifying any issues or bottlenecks that could impact system performance, before they escalate into critical problems. However, the challenge lies in the fact that these tools may not be entirely suitable or effective when applied to microservices that require low latency.

In our scenario, we are dealing specifically with low-level C microservices that communi-

cate with each other via ZeroMQ sockets. The objective is to gather TD across all these microservices.

One crucial aspect to consider is the subsequent analysis of this collected data. While live monitoring (the ability to observe the system in real-time while it operates in a production environment) is desirable, it is not a necessity. A post-mortem analysis approach is acceptable.

We also need to consider the characteristics of the hosts running our microservices. Often, these hosts are equipped with limited hard drive storage, necessitating the implementation of a filtering mechanism to minimize the volume of data saved on the disk. For instance, we should have the flexibility to determine at runtime if we want to save traces of heartbeat signals, particularly for distributed services that frequently transmit synchronization messages. Furthermore, many applications are run on hosts with constrained resources. The installation of any data collection agent or OB could significantly impact the application behavior.

Based on the requirements provided, the following elements of the problem can be identified:

Efficient TD collection The challenge lies in efficiently gathering TD while managing the limited resources of our microservices hosts.

Impact on system performance The process of data collection could itself impact the system performance. The challenge is to minimize this impact while ensuring the necessary data for monitoring and optimization is collected.

Network traffic minimization TD collection could involve significant network traffic. The problem here is to reduce the amount of data sent across the network, which could potentially affect the system performance.

Structured data collection For effective analysis, the TD must be collected and stored in a structured manner. The problem here is to ensure structured data collection that facilitates subsequent analysis.

Data analysis across multiple hosts By examining the TD collected from various hosts, we can identify performance bottlenecks. The challenge lies in suggesting techniques for this analysis.

1.3 Research objectives

The work presented in this thesis seeks to address the following research question: Which tools and strategies are most appropriate for the efficient collection of TD from resource-limited, low-latency microservices?

We aim to accomplish the following specific objectives:

1. Develop a prototype of a system composed of low-latency microservices in C communicating with each other using ZeroMQ sockets.
2. Formulate and benchmark a strategy incorporating a range of tracing tools suitable for collecting TD from the aforementioned distributed system.
3. Apply the TD collection strategy within the system.
4. Develop a strategy that uses a variety of tools for visualizing and analyzing TD.
5. Apply the analysis strategy to the gathered data.

1.4 Outline of the research

Chapter 2 serves as a critical review of the literature, presenting the state-of-the-art tools used to trace and analyze distributed systems and low-latency applications.

Chapter 3 describes the methodology that allowed us to design, test, and evaluate the solution proposed in this thesis.

Chapter 4 forms the core of this thesis. It is an article that describes the proposed solution and discusses the results obtained.

Chapter 5 complements and deepens the discussion presented in the article.

Chapter 6 concludes this thesis by summarizing the work carried out.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we aim to showcase the prior work conducted in domains relevant to our study. Our initial focus will be on the cutting-edge tracers and methodologies employed for the collection of TD. Subsequently, we will explore the techniques and tools used for the analysis.

2.1 Distributed tracers

Distributed tracing is a method used for profiling and monitoring applications, especially those built using a microservices architecture. It helps in understanding how individual services in a distributed system interact to serve user requests, thereby making it a crucial tool for identifying performance bottlenecks, latency issues, and other problems in complex, distributed systems [10]. The concept of distributed tracing was popularized by **Dapper** [11], a system developed by Google. Introduced in a 2010 paper, **Dapper** was one of the first large-scale distributed tracing systems. **Dapper**'s design influenced many open-source distributed tracing systems that followed, such as **Zipkin** and **Jaeger**, and it helped establish distributed tracing as an essential component of observability in modern software systems. In the following, we will undertake an exploratory journey through the evolution of distributed tracing, highlighting all the innovative tools that have been developed along the way.

Pinpoint (2002)

Pinpoint [12] was an innovative research prototype developed in 2002 by teams at the University of California, Berkeley, and Stanford University. It aimed to identify the root causes of issues in internet services, marking a significant shift in problem-solving approaches.

The prototype's genesis was in response to the growing disparity between traditional problem-solving techniques and the emerging dynamic, constantly evolving internet services. The conventional method of root cause analysis involved static system modeling, which proved increasingly impractical for maintaining up-to-date and accurate models in such a rapidly changing environment.

Pinpoint was a pioneer in advocating for a data-driven, request-oriented approach. Instead of proactive system modeling, the researchers proposed using system measurements and recorded data for post-issue troubleshooting. Central to **Pinpoint** was the ability to group information on a per-request basis. This was achieved by assigning a unique request ID to

each request and maintaining it in thread-local storage, a concept that has evolved into the trace contexts we see today.

However, **Pinpoint** scope was limited to a single-machine Java Enterprise environment, and it did not explore context propagation between machines or user-created threads.

Magpie (2004)

Magpie [13], developed by Microsoft Research Cambridge in 2004, was an industrial research prototype with the objective of recording detailed end-to-end traces, similar to current tools like **Jaeger** or **Zipkin**. Unlike **Pinpoint**, it was designed for arbitrary, heterogeneous .NET applications and aimed to annotate traces with detailed information about resource consumption, such as I/O and CPU measurements.

The technical focus of **Magpie** was to disentangle concurrent requests executing within the same software components. However, it confronted a challenge that **Pinpoint** had avoided: the difficulty of propagating request IDs in a heterogeneous environment, without ubiquitous middleware. The solution was to infer correlations mostly from existing outputs, as opposed to exhaustive source code instrumentation.

Magpie integrated with Windows XP Event Tracing for Windows (ETW), a lightweight event logging framework. It inferred relationships between events and added additional ones in specific places to complete the end-to-end picture of a request. This approach relied on a developer-supplied scheme for parsing events and extracting correlation IDs.

Despite its merits, this inference-based approach was less scalable and more brittle, as it depended on event parsing and correct developer-supplied schemas. The industry has since decided that the effort of context propagation in distributed tracing is worthwhile.

Nonetheless, **Magpie**'s significant contribution was demonstrating the value of incorporating fine-grained resource usage information into traces, offering a model for enhancing the value of other information sources to augment traces in modern distributed tracing.

X-Trace (2007)

X-Trace [14], a tracing framework developed in 2007 by researchers at the University of California at Berkeley, aimed at troubleshooting requests executed across multiple machines, layers, and administrative domains. It emerged during a time of increasing complexity and heterogeneity in internet services, where the need for request-oriented tracing was growing. Unlike earlier tools like **Pinpoint** and **Magpie**, **X-Trace** aimed for generality and minimal

user requirements in a heterogeneous environment.

Two significant design choices were made: a standard for context propagation to coherently combine information recorded in different components, and out-of-band report collection to separate information recording from its usage. **X-Trace** introduced the concept of including a dynamic parent ID along with a request ID, allowing deterministic reconstruction of the order and concurrency of events. This eliminated the need for post-processing to infer relationships based on timing or system internals.

The separation of information recording from its usage offered flexibility to developers, and allowed administrators to control their portion of the trace data. Current distributed tracing frameworks follow the **X-Trace** philosophy, focusing on capturing minimal necessary pieces for trace construction, thus facilitating use in highly heterogeneous systems. **X-Trace**'s parent ID concept is analogous to the parent span IDs used in contemporary distributed tracing frameworks.

Dapper (2010)

Dapper [11], an internal tracing framework developed at Google, was first introduced in a 2010 whitepaper and continues to be in use today. Its design serves as the foundation for many of today's popular distributed tracing frameworks. When **Dapper** was created, Google internal systems were based on Remote Procedure Calls (RPC), similar to today's microservice architectures. These systems spurred the need for a distributed tracing framework that could provide request-oriented visibility into large, diverse production systems.

Dapper built upon concepts from **Pinpoint**, **Magpie**, and **X-Trace**, but also confronted fresh operational challenges. It introduced the span model of tracing, a significant shift from the event-based approach of previous systems. Instead of instantaneous points in time, or events, **Dapper**'s spans represented well-defined segments of request execution, providing immediate timing information about the most crucial parts of a request. This was key to diagnosing performance problems, particularly those related to request latency.

Although previous works such as **X-Trace** argued that distributed tracing should impose a minimal set of requirements, **Dapper** imposed an additional requirement on its users by incorporating spans as a first-class concept, thereby enhancing the usefulness of the resulting traces. As the first publicly described distributed tracing framework used by a major company, **Dapper** also revealed operational challenges not previously considered, including trace sampling, runtime overhead trade-offs, security issues, and how to make trace data accessible to users.

Zipkin (2012)

Zipkin [15] is an open-source distributed tracing system that was developed at Twitter. It was inspired by Google Dapper and was created to meet the need for a distributed tracing system in Twitter's complex, distributed architecture. Just like Dapper, Zipkin provides the ability to trace requests and responses across multiple systems, but it does so in a way designed to be easy to adopt in existing systems.

Twitter open-sourced Zipkin in 2012 [16] so that other companies and developers could benefit from the same tools that helped them understand their distributed systems. It provides a way to instrument your system, collect trace data, and analyze that data with a web-based UI. This makes it easier to understand how a system is behaving, where bottlenecks are, and where performance could be improved.

Zipkin was designed to be easy to implement and use. It has a simple architecture and can be integrated into various programming languages and systems. The core parts of Zipkin include the collector, storage, search, and web UI. The collector receives trace data and sends it to storage. The storage then stores the data in a way that can be searched and retrieved. The search component allows users to find specific traces based on different criteria, and the web UI provides a visual representation of the traces.

Opentracing (2016)

The OpenTracing specification [17] emerged as a collaborative solution to the problem of incompatible APIs and data models across various distributed tracing tools. In the early days of distributed tracing, systems like Zipkin had their own unique APIs, making it difficult for developers to switch between tools or use multiple tools together.

Recognizing this challenge, several leading technology companies, came together to define a shared, vendor-neutral, standard API for distributed tracing systems. The result was the OpenTracing specification, which was announced by the CNCF in 2016 [18].

The goal of OpenTracing was to provide developers with a consistent, flexible, and easy-to-use API that could work across different tracing systems. This meant that a developer could instrument their code once, using the OpenTracing API, and could then choose to use any compatible tracing system at runtime [19].

The OpenTracing specification has continued to evolve and expand over the years, driven by an active community of contributors. It provides reference API implementations for several different programming languages, enabling consistent instrumentation across polyglot systems.

As of January 2022, the **OpenTracing** project was deprecated and archived by the CNCF in favor of its successor, **OpenTelemetry** [20].

Jaeger (2017)

Jaeger [21] is an open-source, end-to-end distributed tracing system that was developed by Uber Technologies to handle Uber’s complex architecture and high-throughput requirements. It was designed to help developers monitor and troubleshoot complex, microservices-based architectures.

Jaeger was conceived to address specific needs not met by **Zipkin** or other existing systems at the time [22]. While retaining **Zipkin**’s foundational principles, **Jaeger** introduced unique features like high-resolution, low-overhead sampling, distributed context propagation, and advanced storage and visualization capabilities. **Jaeger** was also built to be compatible with the **OpenTracing** API, from the start, to ensure that it could seamlessly interact with other systems using the standard.

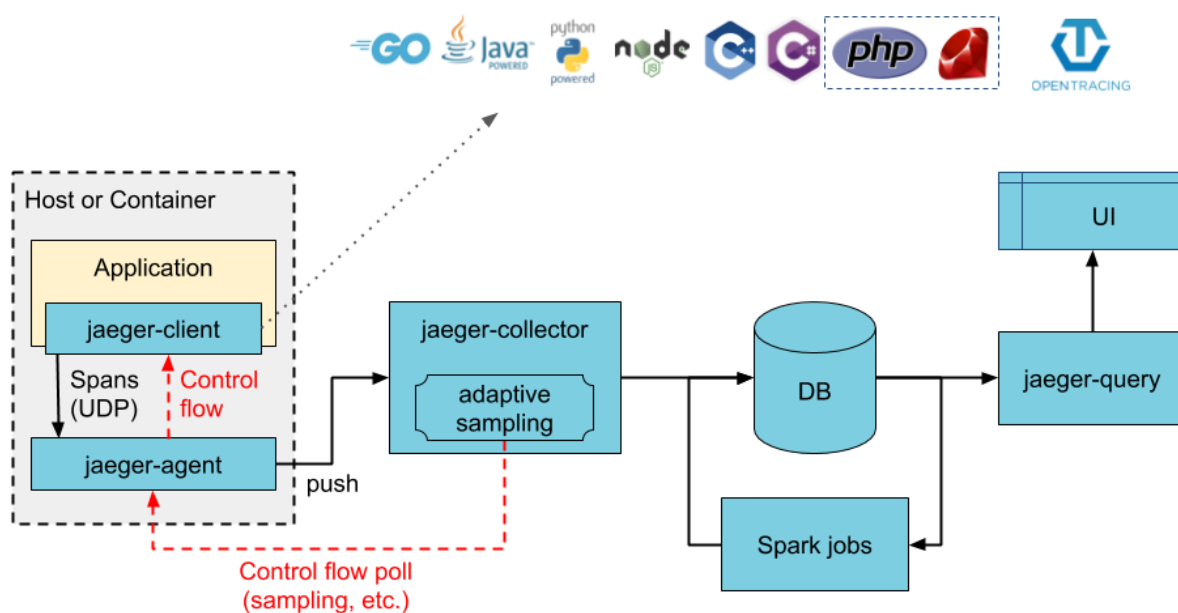
Initially, all **Jaeger** tracing system was made out of 5 components that could be run in a single process (Figure 2.1a), or as a scalable distributed system (Figure 2.1b) [23]:

Client Clients are language-specific implementations of the **OpenTracing** API, used to manually instrument applications or integrate them with various open-source frameworks, such as Flask and gRPC. These instrumented services generate spans for incoming requests and append context information to outgoing requests. However, only the context information is propagated with the requests, while additional profiling data is sent asynchronously to the **Jaeger** backend.

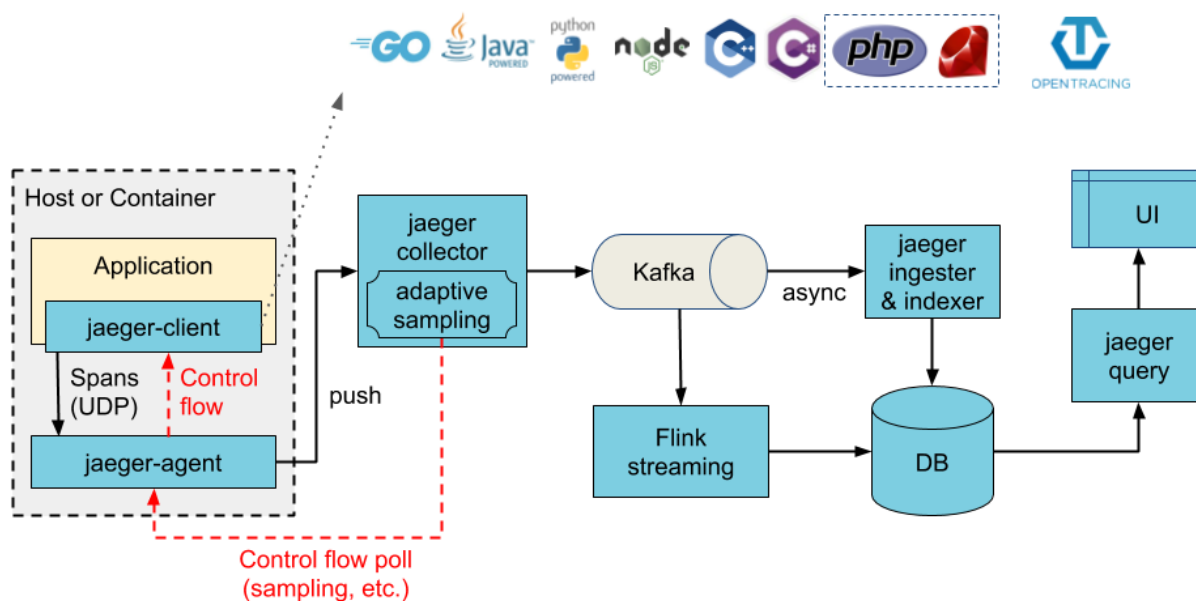
The instrumentation is designed to be always active in production with minimal overhead, achieved through various sampling strategies employed by **Jaeger** clients. These strategies determine when to capture and transmit profiling data. By default, **Jaeger** clients sample 0.1% of traces, but they can also retrieve different sampling strategies from the **Jaeger** backend for more tailored profiling.

Agent The agent is a network service that operates as an intermediary between the client and the collector. It receives spans transmitted over UDP, batches them, and forwards them to the collector. It is designed to be deployed to all hosts as an infrastructure component. The agent abstracts the routing and discovery of the collectors away from the client.

Collector The collector, which obtains traces from **Jaeger** agents, operates a processing



(a) Jaeger with a direct-to-storage architecture: Collectors are writing directly to storage.



(b) Jaeger architecture with Kafka as intermediate buffer: Collectors are writing to Kafka as a preliminary buffer.

Figure 2.1 Jaeger deployments options - Image source [23]

pipeline that validates, indexes and transforms the traces before storing them. **Jaeger**'s storage component is adaptable and currently compatible with Cassandra, Elasticsearch, and Kafka.

Query service The Query service retrieves traces from storage, and hosts a UI to display them.

Ingestor The Ingestor is a service that reads from Kafka *topics* and writes to another storage backend (Cassandra, Elasticsearch).

With the deprecation of the **OpenTracing** project in favor of **OpenTelemetry** (OTel), **Jaeger** has adapted accordingly. **Jaeger** now supports the OTel Collector as a native component, acting as a receiver and exporter within the OTel ecosystem [24]. **Jaeger** clients and the agent have been deprecated respectively in favor of OTel instrumentation API/SDKs and OTel collector. This allows users to continue leveraging **Jaeger**'s powerful tracing capabilities, while benefiting from the comprehensive observability framework provided by OTel.

OpenCensus (2018)

OpenCensus [25] was an open-source project that aimed to make the collection and submission of app metrics and traces easier for developers. It was initially developed by Google and provided libraries for multiple programming languages to capture, manipulate, and export metrics and traces in a standardized manner.

OpenCensus was designed to work with various back-end services, enabling developers to choose where they wanted to store and analyze their data. Key features included automatic context propagation, various sampling strategies, and APIs for custom metric and trace collection [26].

However, in a bid to avoid fragmentation and duplication in the telemetry and observability space, **OpenCensus** and **OpenTracing**, decided to merge [27]. The result is OpenTelemetry, a new, unified standard for observability telemetry. This merged project, now part of the CNCF, aims to provide a comprehensive set of APIs, libraries, agents, and instrumentation for capturing traces and metrics from applications.

OpenTelemetry (2019)

OpenTelemetry (OTel) [28] is an open-source observability framework that generates, collects, and manages TD from software services. Crucially, OTel is vendor- and tool-agnostic,

meaning that it can be used with a broad variety of OBs, including open-source tools, as well as commercial offerings.

The **OTel** project was first announced in May 2019 as a merger of the **OpenTracing** and **OpenCensus** projects to avoid duplication of effort and to create a standardized approach to observability [27]. The first official release of the **OTel** specification was made in November 2020. **OTel** is also a project within the CNCF.

OTel consists of the following major components:

- A specification for all components.
- A standard protocol that defines the shape of TD.
- Semantic conventions that define a standard naming scheme for common TD types.
- APIs that define how to generate TD.
- Language SDKs that implement the specification, APIs, and export of TD.
- A library ecosystem that implements instrumentation for common libraries and frameworks.
- Automatic instrumentation components that generate TD without requiring code changes.
- The **OTel** Collector, a proxy that receives, processes, and exports TD.
- Various other tools, such as the **OTel** Operator for Kubernetes, **OTel** Helm Charts, and community assets for FaaS (Function-as-a-Service).

OTel specification The specification [29] defines standard procedures and conventions for collecting, processing, and managing TD. This specification serves as a guide for all APIs, SDKs, and protocol implementations.

The specification outlines the key concepts and terminologies used in **OTel** and provides a comprehensive framework for capturing and managing TD. It defines the structure and format of traces and metrics, as well as the protocols for sending them to other systems.

It also outlines standards for context propagation, which allows for the correlation of telemetry across multiple distributed systems. Other conventions covered by the specification include resource identification, semantic conventions for common scenarios, and error handling.

The **OTel** specification is continually updated and refined by a diverse community of contributors to reflect the evolving needs of developers and the broader technology landscape. Its goal is to provide a consistent and unified approach to observability across all programming languages and platforms.

OTel protocol (OTLP) OTLP [30] is a general-purpose TD delivery protocol designed in the scope of the **OTel** project. Its specification describes the encoding, transport, and delivery mechanism of TD between telemetry sources, intermediate nodes such as collectors, and OBs.

The **OTel** Protocol (OTLP) operates on a request-response model. The client sends a request to the server and waits for a response. The client can be any application or service that produces TD, such as traces, metrics, and logs. Typically, clients are web servers, microservices, or any instrumented software that generates TD.

The server, on the other hand, would be the observability backend or any platform that receives, processes, and possibly visualizes this TD. This could be a service like **Prometheus**, **Jaeger**, or any application that is built to handle the ingestion and analysis of TD.

OTel Protocol (OTLP) uses Protocol Buffers (protobuf) as its data format and gRPC/HTTP for data transmission, providing advantages like efficient serialization and strong type checking. Protobuf provides language-neutral and platform-neutral schemas, allowing developers to easily define how data should be structured. This universal language support is advantageous because it enables OTLP to be easily implemented across various languages. Developers can leverage protobuf's generated APIs, which are available in numerous languages, to serialize the structured data (TD in this case) to protobuf format and deserialize from protobuf format to the data structures in their programming language. This facilitates the transmission of TD using OTLP. The appendix A gives an example of a request trace and request trace gRPC export definitions.

Semantic conventions Semantic Conventions [31] are agreed-upon standards for how to record specific types of data, like HTTP requests, database calls, or container information. They define a common set of attributes, their types, and their meanings to be used when recording TD. This ensures consistency and uniformity of data across different services, languages, and backend observability platforms, making the data more useful and interpretable. The appendix B gives an example of semantic conventions for servers general attributes.

APIs APIs [32–37] define how to generate TD. These APIs are language-specific and pro-

vided through Software Development Kits (SDKs) for various popular languages like Java, Python, JavaScript, Go, etc. Key **OTel** APIs include the Tracing API for distributed request trace collection, the Metrics API for numerical data capture, the Baggage API for context propagation across process boundaries, the Context API for context management, and the Resource API for describing the running compute unit.

SDKs While APIs are used to instrument the code and generate TD, the SDKs [38–41] are responsible for processing and exporting that data. SDKs are the implementations of the APIs. They are language-specific and provide the tools needed to capture, process, and export TD from your services to the backend observability platforms. The SDKs include components for tracing and metrics and handle tasks such as span processing, metrics aggregation, and data exporting. They also provide additional features like context propagation, batching, retry mechanisms, and configurable sampling strategies.

Library ecosystem and automatic instrumentation The library ecosystem offers a wide range of language-specific libraries that provide automatic instrumentation for many popular frameworks and libraries [42]. These libraries handle the task of injecting the necessary telemetry collection code into your application, reducing the manual effort needed to instrument your code.

There are libraries for Java, Python, JavaScript, Go, .NET, and more languages. Within each of these languages, there are specific libraries for instrumenting popular frameworks and libraries. For instance, in Java, there is instrumentation for frameworks/libraries like Spring, gRPC, JDBC, and many others. Similarly, for Python, there is instrumentation for Django, Flask, SQLAlchemy, and more.

These instrumentations capture relevant traces, metrics, and context automatically, when services using these libraries or frameworks are run, making it easier to adopt **OTel** in projects.

Collector The Collector [43] is a component that can receive, process, and export TD. It is designed to be a vendor-agnostic implementation and can be used as an agent (running alongside the application) or as a standalone service.

Key functions of the Collector are shown in Figure 2.2. Initially, the collector receives TD from applications using multiple protocols, including OTLP, **Jaeger**, **Zipkin**, and others. Next, it processes the received data, which can involve transforming, aggregating, or enhancing the data with additional metadata, or even reducing it through sampling. Finally, it exports the processed data to one or more observability backends, once again supporting multiple protocols. Moreover, the Collector supports extensions,

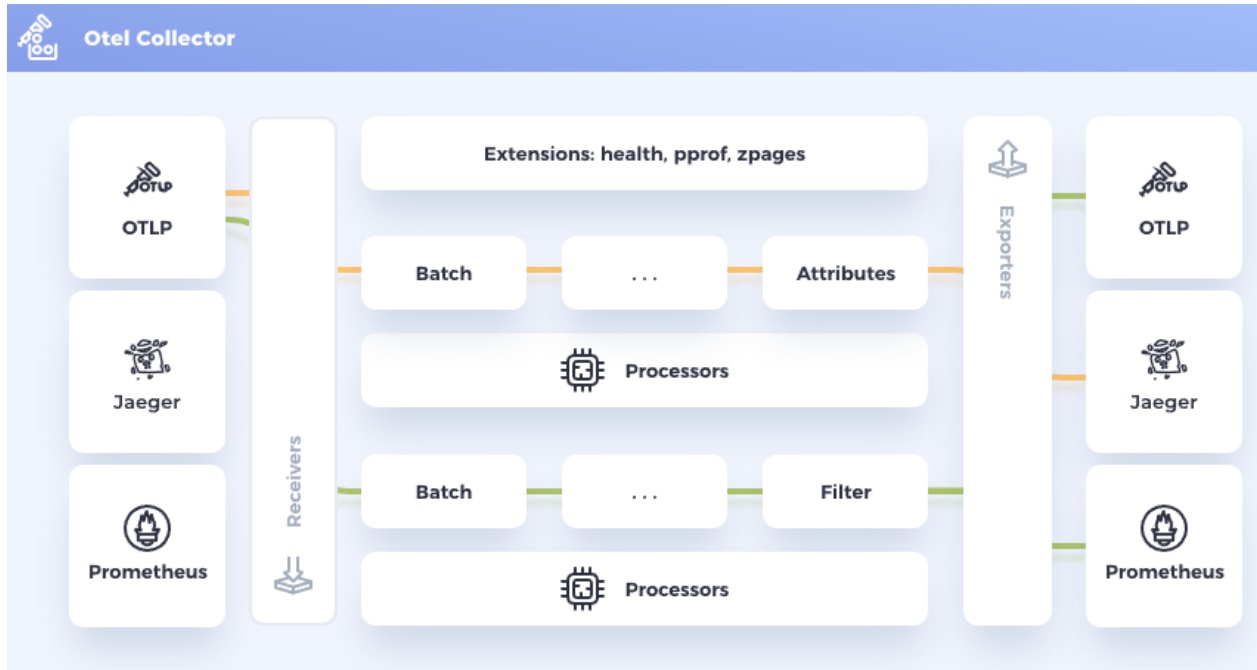


Figure 2.2 **OpenTelemetry collector features** - Image source [43]

which are not part of the data path but provide supporting capabilities, like health checks and diagnostics.

2.2 Kernel and userspace tracers

Ftrace

The **Ftrace** [44] tool is present alongside the kernel in Linux systems since version 2.6.27 and allows kernel events tracing. Although **Ftrace** is named after the function tracer it also includes many additional functionalities.

It relies on static tracepoints inserted into the Linux kernel by developers, and also allows the user to provide their own tracepoints through dynamically inserted probes, known as **kprobes**. It is worth noting that these static tracepoints inserted into the kernel are in the form of a **TRACE_EVENT** macro, which is shared with other kernel tracers. Similar to the control groups used by Linux containers, which can be controlled from the **cgroup** file system exposed by the kernel, **Ftrace** can be controlled from the **tracefs** file system [45].

While the tracer latency is fairly good when a tracepoint is reached, Gebai and M.R. Dagenais have observed that the performance of the tracer tends to degrade on multicore systems, which are common in cloud infrastructures [46].

LTTng

Since taking over from its predecessor, **LTT** (Linux Trace Toolkit), **LTTng** (Linux Trace Toolkit: next-generation) [47] has established itself as a reference tracer for Linux. **LTTng** utilizes static tracepoints embedded within the kernel to capture events. In addition to this, it can gather event data from dynamic tracepoints supplied through the **kprobes** mechanism, providing an extensive and flexible tracing capability.

LTTng is not included as a part of the mainline Linux kernel. Instead, it is deployed as a group of loadable kernel modules, specifically for the kernel tracing and trace consumption components. This unique deployment method allows for a different approach to trace consumption, as opposed to the **tracefs** interface of **Ftrace**. **LTTng** provides a more flexible and customizable tracing experience, giving users the freedom to load only the necessary modules for their specific tracing tasks. This modular approach not only reduces the resource load on the system, but also allows for independent updates and enhancements to **LTTng**, without impacting the core kernel.

LTTng offers several key advantages that make it an ideal choice for system analysis and debugging work:

Efficient design that minimizes tracing overhead **LTTng** is designed to have a minimal impact on performance [48] while being fully reentrant, interrupt-safe, and non-maskable interrupt-safe. Unlike tools like **eBPF** and **SystemTap**, which focus on real-time monitoring and calculation of relevant metrics, the goal of **LTTng** is to collect very low-level events, accurately and comprehensively, for offline analysis.

LTTng can handle a high volume of events, especially when the system load is high. It has a reasonable performance overhead and barely affects the system behavior during its observation. This is a significant advantage when debugging problems that rely on finely-tuned event scheduling, which could be altered simply by the tracer’s presence.

LTTng collects kernel events with a high precision, which is crucial for detailed system analysis. While only microsecond precision is available on many tracers, **LTTng** is capable of capturing these events on a nanosecond scale.

According to a 2018 comparison of tracers by M. Gebai and M. R. Dagenais [46], **LTTng** is the most efficient kernel tracer for Linux. A key element in achieving this efficiency is the use of per-CPU data structures, local atomic operations as main buffer synchronization primitive, and RCU (Read-Copy Update) mechanism to control tracing [49]. This design allows concurrent recording of events from multiple CPUs, without the need for locking mechanisms, which could slow down the tracing process. This way,

data is written into these buffers when an event occurs, and then it is persisted to disk when system resources allow for it.

Unless in blocking mode, LTTng developers have prioritized system performance, opting to maintain system behavior, even if it means losing some trace events. This can happen when the event rate is too high to persist the events to disk in a timely manner. However, thanks to the per-CPU ring buffers, LTTng can manage high event rates more efficiently, ensuring minimal impact on the system performance [50].

Fine-grained control over tracing activities LTTng provides fine-grained control over tracing activities, which is one of its most powerful features. This enables users to specify exactly what events to trace, thereby reducing system overhead and increasing performance.

This fine-grained control is achieved through the use of tracing sessions, channels, and events. Users can create a tracing session and then define channels within that session. Each channel can be configured with different parameters, such as buffer size, and can have multiple events associated with it. Users can then enable or disable these events individually or in groups, providing precise control over what data is collected.

In addition, LTTng supports both inclusion and exclusion filters for events. This allows users to specify conditions for when an event should be traced, further refining the control over tracing activities.

Overall, this level of control allows users to focus on the specific areas of interest in their system, reducing the amount of irrelevant data collected and making the analysis of trace data more manageable and efficient [50].

Fast to write trace format LTTng uses the Common Trace Format (CTF) [51] to store trace data. This format is designed to be fast to write, making it ideal for capturing large amounts of data in real time. The use of CTF in LTTng contributes to its high performance, ensuring that tracing activities have minimal impact on system performance, while providing rich detailed data for analysis.

CTF is a binary format, which is more compact and faster to write compared to text-based formats. It has a low overhead, which is crucial for maintaining system performance during tracing. The format also has a flexible design that can accommodate different types of data, including complex data structures, making it suitable for a wide range of tracing applications.

Additionally, CTF is designed to be easily readable by machines, facilitating efficient data analysis. It includes metadata that describes the data structures used in the trace,

allowing trace analysis tools to interpret the data correctly without additional input. This makes CTF not only fast to write, but also efficient to read and process, which is crucial for effective system analysis.

Remote tracing LTTng supports remote tracing, a feature that allows tracing of applications running on a different machine than the one on which the LTTng session is running. This is particularly useful for distributed systems where applications and services are spread across multiple machines.

To enable remote tracing, LTTng uses a client-server model. The LTTng client (on the machine where the session is started) communicates with the LTTng session daemon (`lttng-sessiond`) running on the target machine (the machine to be traced). Figure 2.3 illustrates the trace data path.

The client can start and stop tracing, enable and disable events, and configure other tracing parameters on the target machine. The trace data collected from the target machine can then be retrieved and analyzed on the server machine.

This ability to perform remote tracing makes LTTng a versatile tool for analyzing and debugging distributed systems and cloud-based applications.

Compatibility with various trace analysis tools By producing trace data in the Common Trace Format (CTF), LTTng allows for easy integration with a wide range of analysis tools that support this format.

One of the primary tools used for analyzing LTTng trace data is the **Trace Compass**, an open-source application that provides a comprehensive set of features for visualizing and analyzing traces. LTTng traces can also be analyzed using the Babeltrace command-line tool, which supports conversion between different trace formats, including CTF.

Moreover, LTTng's compatibility extends to scripting languages such as Python, allowing users to write custom scripts for trace analysis. This opens up possibilities for automated analysis and allows users to tailor their analysis workflows to suit their specific needs.

Active development LTTng is the result of more than 10 years of active open-source development by a community of passionate developers. LTTng is currently available on major desktop and server Linux distributions.

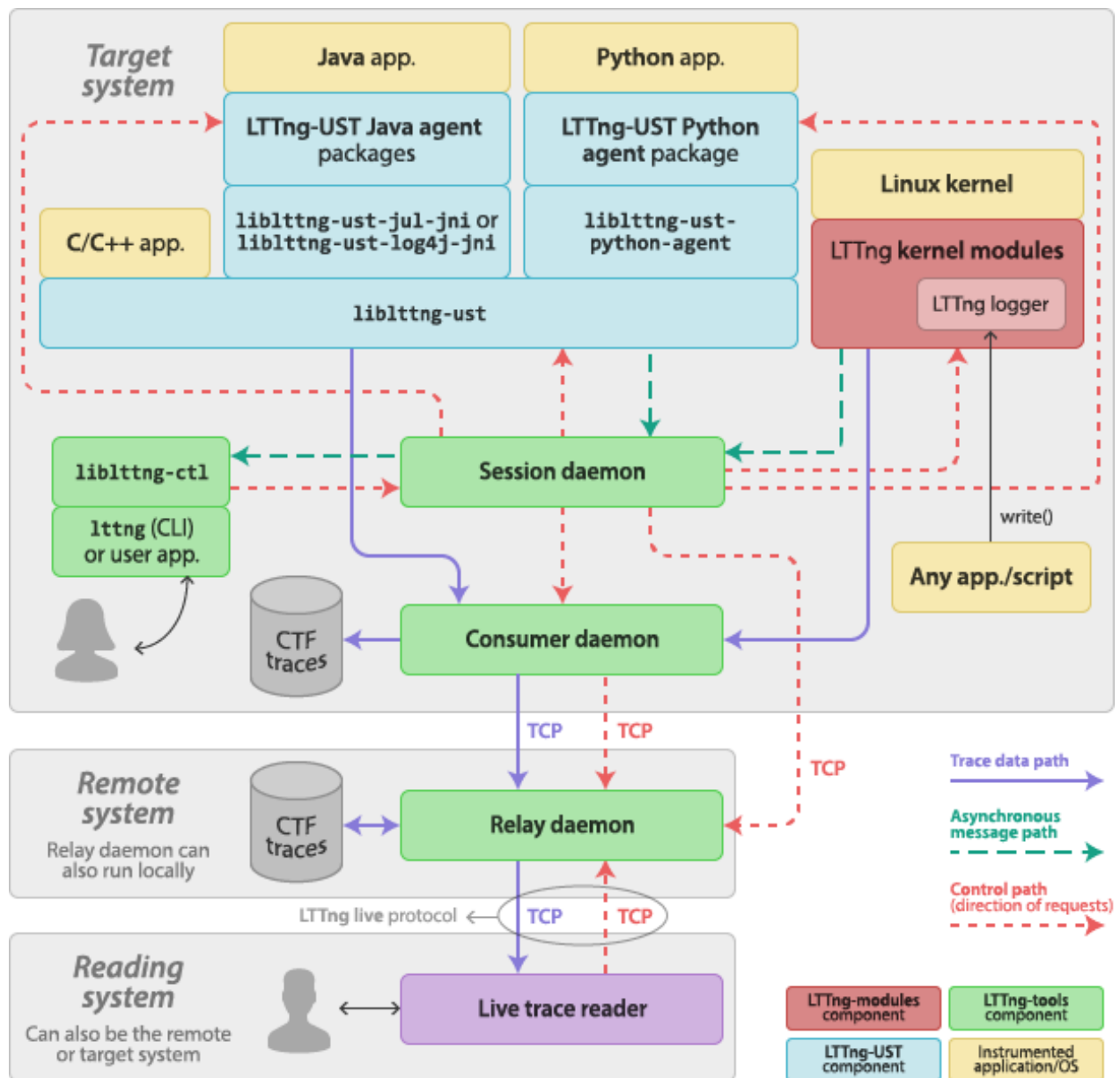


Figure 2.3 **Components of LTTng** - `lttng-ust-*` libraries are utilized for user-space tracing, `lttng-modules` are kernel modules enabling kernel event tracing, and `lttng-tools` comprise a suite of command-line tools for session and event management, inclusive of the `lttng` command for tracing control and `lttng-relayd` for network streaming. - Image source [50]

LTTng-UST

LTTng-UST (LTTng User Space Tracer) [52] is a key component of the LTTng suite. As the name suggests, it is designed to trace applications running in user space, complementing LTTng’s kernel tracer.

Unlike the kernel tracer, which collects data about system-wide events like system calls and kernel operations, LTTng-UST is used to trace events specific to a particular application. This can include function calls, variable values, or any other custom events defined by the developer.

LTTng-UST uses a similar design to the kernel tracer, including the use of ring buffers for efficient data storage and the CTF for trace data. It also supports the same fine-grained control over tracing activities, allowing users to specify exactly what events to trace.

LTTng-UST empowers developers to create customized tracepoints within their code. This feature enables the capture of detailed, application-specific information at any given point. The data collected can range from function calls to any other relevant application-specific data. Similar to kernel tracepoints, these user-defined tracepoints can be enabled or disabled at runtime as required, offering a flexible approach to application tracing.

Perf

Perf [53] is a performance monitoring tool integrated into the Linux kernel. It is typically used for sampling and profiling applications, differing from the use cases of **Ftrace** or **LTTng**. However, it can also interface with the tracepoint infrastructure within the kernel to record tracepoints, including system calls.

Perf can gather hardware-level Performance Monitoring Unit (PMU) metrics such as cache misses, Translation Lookaside Buffer (TLB) misses, CPU cycles, missed branch predictions, and more. **Perf**’s monitoring scope can be restrained to a single process [54], making it particularly suited for analyzing the behavior of a specific process. It can help answer practical questions about the code’s cache-friendliness or the amount of time spent in each function.

The **Perf** tool is built on top of the kernel `perf_events` subsystem, which implements the tracing, profiling, and sampling functionalities [55]. **Perf** can also use the **Ftrace** architecture to hook to tracepoints and trace, similarly to **Ftrace** and **LTTng**.

eBPF

eBPF [56], which stands for Extended Berkeley Packet Filter, has evolved from the Berkeley Packet Filter (BPF) and is now an independent monitoring tool included in the Linux kernel. It allows users to write programs, similar to probes, that can be dynamically inserted at any location within the kernel using **kprobes**.

eBPF programs are compiled into bytecode, which is then interpreted and executed by the kernel, in a restricted context to maintain security. The kernel also supports just-in-time compilation for parts of the generated bytecode.

It is important to highlight that **eBPF** is not a tracer in the traditional sense, as it does not follow the callback, serialize, write scheme [46]. Instead, it serves as an aggregator. The **eBPF** interpreter provides user-friendly data structures, such as simple arrays and hashmaps, making it an excellent tool for aggregation, bookkeeping, and real-time monitoring. It provides arrays that can be shared from the kernel space to the user space, enabling a tracer-like behavior to be implemented.

SystemTap

Similar to **eBPF**, **SystemTap** [57] offers a straightforward scripting language that allows users to write probes for real-time monitoring and data aggregation. It enables users to create bespoke probes to observe their systems by providing kernel or userspace symbols to connect to.

SystemTap scripts are converted into C code and compiled into a module that can be loaded into the kernel. Once this module is loaded, it uses **kprobes** to dynamically insert the probe into the kernel code.

Strace

Strace [58] is a utility designed for tracing system calls. **Strace** operates by utilizing the **ptrace** system call. It hooks into a process and intercepts all its system calls, including the associated arguments. The collected data is then recorded to a file descriptor, which can be examined later for analysis.

Strace can impose a substantial overhead due to its intensive trap mechanism and scheduling costs. This is because it involves multiple processes - the one being monitored and the one doing the monitoring. As a result, **Strace** may not be ideally suited for production environments, due to this added overhead [46].

2.3 Observability agents

Observability agents are software components installed on a host or in an application that collect and send TD to a central observability or monitoring system. Two observability agents were already discussed in this document: the **OTel Collector** and the **Jaeger Agent**. Here are more examples of observability agents. **Prometheus Node Exporter** [59] is an agent for the **Prometheus** monitoring system. It collects system-level metrics such as CPU usage, memory, I/O, etc., from the host machine and exposes them for scraping by the Prometheus server. **Fluentd** and **Fluent Bit** [60] are open-source data collectors for building observability pipelines. They can collect logs from various sources and send them to multiple destinations. **Elastic Beats** [61] are lightweight data shippers that you can install on your servers to send operational data to Elasticsearch or Logstash. They can collect various types of data - logs (Filebeat), network data (Packetbeat), metrics (Metricbeat), and more.

In addition to the previously mentioned agents, almost all cloud-based observability services offer their own special agents. These agents are designed to gather specific TD and transmit it to their respective cloud observability service. For example, **The New Relic Agents** [62] gather application performance data such as transaction times, error rates, and slow SQL queries. The **AWS CloudWatch Agent** [63] collects system and application metrics from Amazon EC2 and on-premises servers for CloudWatch analysis. **Google Cloud Operations Agent** [64], previously known as the Stackdriver Agent, collects monitoring and logging data from Google Cloud, AWS, and hybrid environments. **Azure Monitor Agent** [65] accumulates TD from various sources and forwards it to Azure Monitor or Azure Log Analytics for detailed analysis.

Despite the differences in their specifics and the data they collect, these observability agents share a common internal design structure:

Data Collection All agents are designed to collect various types of data. The data collected could be system-level metrics (like in **Prometheus Node Exporter**), application logs (Fluentd, Fluent Bit), operational data (Elastic Beats), or other specific data (New Relic Agents, AWS CloudWatch Agent, Google Cloud Operations Agent, Azure Monitor Agent). The type of data they collect depends on their design and purpose. For system metrics, this could involve reading from specific files in the `/proc` or `/sys` filesystems on Linux, using system APIs, or executing system commands and parsing the output. For logs, this can involve reading log files, listening on a network port, or subscribing to a messaging system.

Data Processing After collection, these agents process the data to put it into a format that can be understood by the central monitoring system. This often involves structuring the data and sometimes aggregating it.

Data Transmission The agents are designed to send the collected and processed data to a central monitoring or observability system. This could be a locally hosted system like a `Prometheus` server, or a cloud-based service.

Scalability and Efficiency Observability agents are designed to be lightweight and efficient, so as not to consume significant system resources. They are also designed to be scalable to handle the monitoring needs of large, distributed systems.

Configuration Each agent can be configured to define what data to collect, how often to collect it, where to send it, and other settings. This allows the agent to be tailored to the specific needs of the system it is monitoring.

Compatibility These agents are often designed to be compatible with a wide range of systems and platforms. This includes various operating systems, server types, application languages, and more.

Resiliency Agents are designed to handle failures gracefully. They often have mechanisms to buffer data in case of network issues or if the central system is down.

Security Data collected and transmitted by these agents often have security implications, so they are designed with security considerations in mind. This includes secure data transmission and access controls.

2.4 Telemetry data analysis

In this section, we present tools that are used to receive, process, analyze, or visualize TD, after they have been collected. Our primary focus is on OBs, and then we move on to other analysis tools and techniques.

2.4.1 Observability backends

Request trace databases

The landscape of tools and platforms capable of processing and visualizing distributed request traces is expanding rapidly. Recent surveys [9,66] denoted more than 20 tools and platforms. This includes open-source options such as `Jaeger` [21], `Zipkin` [15], and `SkyWalking` [67], as

well as proprietary solutions like **Datadog** and **New Relic**. Despite many options available, these tools share several common features and functionalities [66]:

Accessing traces A large amount of traces can be stored in a backend database and made accessible through a web interface. Users can locate traces by searching based on high-level, common attributes like a specific service, type of API call, or time window. From the list of matching results, users can choose a specific trace for a more detailed examination.

Swimlane View The “swimlane” or “waterfall” visualization is a common method used by past distributed tracing frameworks like **Dapper** to illustrate individual traces. This approach allows users to conduct a detailed, manual investigation of a single request, which is often a crucial part of troubleshooting tasks.

An example of this type of visualization in **Jaeger** is shown in Figure 2.4. The visualization represents a timeline of a single request, with spans spread out horizontally and sorted vertically. While some visualizations use lines to explicitly show relationships between spans, in Figure 2.4, these relationships are implied based on the vertical order and indentation of span names.

More detailed information, such as annotations and key-value attributes of spans, can usually be accessed by clicking on a particular span - this information is then displayed in a dialog box or side panel. Since an individual trace can be quite large (consisting of thousands of spans and potentially several megabytes in size), there is typically a pan-zoom function for easy navigation.

Alternate views Most of these tools, offer more than just the standard swimlane visualization. For example, **SkyWalking** broadens the visualization options with its List, Tree, and Table views. These varied views can offer distinctive insights, potentially creating a more effective user interaction with the tracing data. This can be particularly beneficial, catering to the unique needs or preferences of the user.

Aggregate visualizations Distributed tracing can gather massive quantities of traces. Facebook reportedly collects over 1 billion traces daily. Some modern systems have started to offer visualizations designed for aggregate analysis. Take **Jaeger**, for example. Its Topology Graphs deliver a comprehensive representation of the service architecture. Meanwhile, **Jaeger**’s Service Performance Monitor View presents performance metrics for each service. These metrics are derived from the latencies and error rates documented within spans. Moreover, **Jaeger** includes a trace comparison feature, enabling

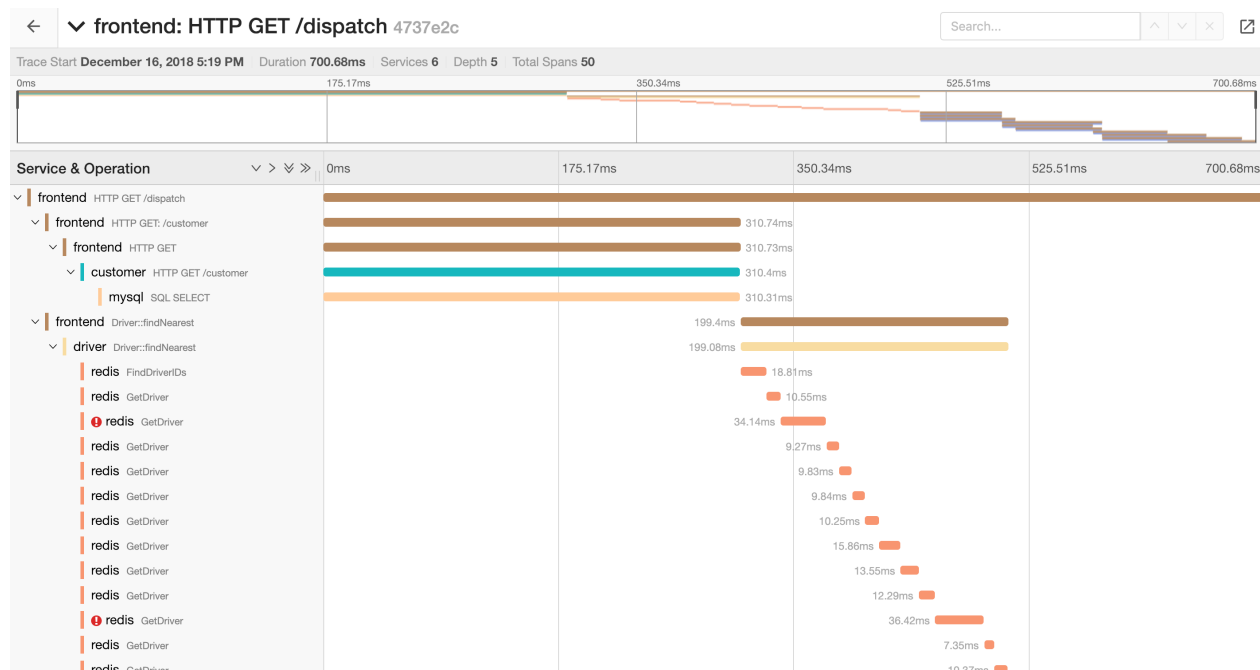


Figure 2.4 **Jaeger** UI showing a sample distributed trace. The view shows the complete end-to-end view of a request journey.

users to juxtapose the structure of a trace with a collection of other traces, for a more in-depth analysis.

Standards Support They typically support key tracing protocol standards like OTLP (cf. section 2.1).

Time series databases

Time series databases (TSDB) are specifically optimized for handling time-series data, i.e., data that is indexed by time. A study conducted in 2016 identified as many as 75 distinct TSDB [68]. Among these, some prominent examples that have gained widespread usage include **Prometheus** [69], **VictoriaMetrics** [70], and **InfluxDB**. Here are some common characteristics that TSDB share:

Data Structure TSDB are built to handle time-series data, i.e., data that is indexed by time. This could be anything from stock market data to sensor data, server metrics, etc.

Scalability They are designed to be scalable and can handle large quantities of data. They are capable of storing and retrieving massive amounts of data in a fast and efficient

manner.

High Performance They are optimized for high insert and query rates, a primary requirement for time-series data.

Time-based operations They provide time-based functions and operations like aggregation over time intervals,

Data Retention and Compression They often include data retention policies and data compression to manage storage over long periods.

Query Language Most of these TSDB provide their own query language or extend an existing query language like SQL to support time-series data.

Resiliency They are designed to be fault-tolerant and highly available, ensuring that data is not lost in case of hardware failures.

Integration with other tools They can integrate with various data collection, monitoring, and visualization tools. For example, Prometheus has built-in alerting and graphing but also integrates with **Grafana** for more complex data visualization.

Prometheus is an open-source system monitoring and alerting toolkit, originally built at SoundCloud. It is now a standalone open-source project part of the CNCF. **Prometheus** is designed around a pull-based model for collecting metrics, as opposed to a push-based model. In a pull-based model, **Prometheus** is configured to scrape metrics from the target systems at regular intervals. This method allows **Prometheus** to control the load, and the frequency, at which it ingests data, making it more resilient to spikes in metrics, which could otherwise overwhelm the system in a push-based model.

While **Prometheus** uses a pull-based model for metrics collection, it also includes a feature called ‘remote write’. This feature allows **Prometheus** to send metrics to a remote storage system. This is especially useful for long-term storage or when you want to aggregate metrics from multiple **Prometheus** servers.

Prometheus is designed for real-time monitoring. It scrapes data from your services at regular intervals, typically every few seconds or minutes, and stores these in a time-series database. **Prometheus** by default automatically assigns a timestamp to every data point at the moment of ingestion/scraping. This timestamp is based on the server’s system time when the metric is ingested.

However, there are cases where you might want to manually set the timestamp for a data point, such as when you are importing historical data or dealing with event-style data where

the event time is significant. While this feature can be useful in certain situations, it is generally recommended to let **Prometheus** handle the timestamping. This is because manual timestamping can lead to issues with metrics reliability and can complicate the setup. For example, if the clocks on your **Prometheus** server and the service exposing the metrics are not perfectly synchronized, you could end up with incorrect timestamps [71]. That being said, **Prometheus** does not provide a built-in way to import data with custom timestamps.

On the other hand, **VictoriaMetrics** is designed to handle both real-time and historical data. It provides a way to import data with custom timestamps, which makes it suitable for storing and analyzing historical data. **VictoriaMetrics** also has better long-term data retention capabilities than **Prometheus**, as it uses advanced compression algorithms to store data more efficiently. **VictoriaMetrics** is a fast, cost-effective, and scalable open-source time series database. It is fully compatible with **Prometheus** and can be used as a drop-in replacement for **Prometheus** for long-term storage, or as a centralized, highly available solution for monitoring multiple **Prometheus** instances. **VictoriaMetrics** supports the **Prometheus** data model, query language PromQL, and the **Prometheus** exposition format. It can scrape **Prometheus** exporters directly, or ingest data from **Prometheus** servers via the remote write API.

Logs backends

Logs are a critical component of any system for debugging, monitoring, and understanding system behavior. Several logging backends can handle, process, and store log data [9]. Some of these include **Loki**, **Elasticsearch**, **Fluentd**, **Graylog**, **Splunk**, **Loggly**, **Papertrail**, **Sumo Logic**, and **Datadog**. All the backends mentioned are designed to handle, process, and store log data. Here are some common features and functionalities they share (not necessarily all):

Log Aggregation They receive/collect logs from various sources and centralize them into one location. This is crucial for managing logs effectively, especially in distributed systems where logs are generated on different servers or services.

Log Processing These backends provide features to process and transform logs, such as parsing, filtering, and enrichment.

Log Storage They provide storage capabilities to retain logs for a certain period of time.

Log Search They offer powerful search capabilities to search through the log data. This helps in troubleshooting and identifying issues quickly.

Real-Time Analysis They provide real-time analytics capabilities, which help in identifying issues as they occur.

Alerting These tools can trigger alerts based on certain conditions or anomalies detected in the logs.

Visualizations and Dashboards They provide features to create visualizations and dashboards from the log data, which help in understanding trends, patterns, and the overall health of the system.

Integration These tools usually offer a wide range of integrations with other tools and services. For example, they can collect logs from various sources, and they can integrate with notification services for alerting.

Scalability These backends are designed to handle large volumes of log data and can scale to fit the needs of the system.

2.4.2 Other tools and techniques

Tracecompass views and analyses

Trace Compass [72] is an open-source application used to visualize and analyze traces in Linux systems. It provides views, graphs, metrics, etc. to help extract useful information from traces, in a way that would be otherwise too hard to process by a human or would take a long time.

Real-time information about a Linux system can be obtained by periodically recording metrics from files located in the `/proc` directory. Tools like `top` and `ps` use this interface, parse the content, and format it for display. This technique provides system statistics at a sampling frequency that is determined by the interface. In contrast, kernel tracing logs all events in a time-ordered manner. Rather than aggregating metric values at pre-defined intervals, kernel tracing allows for the recovery of these values afterward, from the trace, at any desired resolution. This offers a more granular and accurate view of system behavior over time.

Depending on the workload of the targeted system, and the number of active tracepoints, a tracer such as `LTTng` can generate traces of several hundred megabytes of data in just a few seconds of execution. This vast amount of data cannot feasibly be manually analyzed. Furthermore, trace events reveal the low-level behavior of the kernel which requires a deep understanding of kernel internals for analysis. In many instances, the meaning of an event may depend on previous events. Therefore, to extract valuable information from a kernel

trace, fast and reliable analysis tools are necessary. The precise role of trace analysis tools is to scan the traces, based on defined algorithms, and present a synthesis or a more useful view to the user.

A good example of a trace analysis algorithm is the calculation - from only kernel trace events - of higher-level metrics, such as CPU usage rate. Giraldeau et al. [73] propose computing these metrics, with an arbitrary precision chosen during the analysis, from the kernel traces collected on the targeted system. In the case of CPU usage, it suffices to consider only the events of the trace that correspond to a context switch. These events contain the identifier of the process that becomes active, as well as the identifier of the CPU core on which it is scheduled. Thus, for each processor, it is possible to calculate all the intervals that correspond to the execution of a process, and from there, the CPU usage rate over a given interval. Modern trace analysis tools like **Trace Compass** include this type of analysis for kernel traces, and offer an interactive view that the user can browse to identify anomalies present during execution. For example, it is possible to pinpoint at a glance the moment in the trace corresponding to a particularly high total memory allocation, to find the root cause. Some analyses also allow detailing the state changes of a thread over time, as shown in Figure 4.2.

Userspace events can help provide access to application activity unknown to the kernel. Loic Gelle [74] suggests combining distributed tracing with low-level software tracing to identify the root cause of span latencies. The solution involves a hybrid trace collection to capture and synchronize both kernel and distributed request events using **LTTng** and **LTTng-UST**. The distributed tracer is instrumented to add threading information directly into spans, to facilitate postmortem synchronization. As a result, both kernel and distributed traces can be opened in a **Tracecompass** experiment, and thread states can be marked directly on spans.

Grafana

Grafana [75] is an open-source analytics and interactive visualization web application. It provides users with the capability to ingest data from various sources (e.g. **Loki**, **Jaeger**, **Prometheus**, **VictoriaMetrics**, etc ...), query the data, and produce customizable charts for easy analysis. **Grafana** offers a multitude of visualization options to aid users in understanding their data. These visualization options are divided into “panels”, which are then utilized to construct **Grafana** dashboards.

The platform has built-in support for numerous notification channels, including but not limited to email, **Slack**, and **PagerDuty**, thus enabling users to set alerts and be notified of any abnormal activity. **Grafana** also allows users to annotate, or leave notes directly on

graphs, providing a way to conveniently mark significant points on their graphs and serve as a reminder for future actions.

Machine learning and performance anomaly detection

Machine Learning has emerged as a powerful tool for anomaly detection in system performance. This approach leverages the ability of machine learning algorithms to learn from data and make predictions or decisions without being explicitly programmed. In the context of system performance, machine learning can be used to analyze large amounts of system metrics and operational data, learn the normal behavior of the system, and then detect any deviation from this normal behavior as a potential anomaly. This capability is particularly useful in complex systems where it is difficult to define what constitutes normal behavior and where anomalies can have severe consequences. Techniques such as Support Vector Machines (SVM), K-Nearest Neighbors (KNN), Decision Trees, and more advanced methods like Long Short-Term Memory (LSTM) models have been successfully applied in this field. These machine learning techniques have shown significant potential in identifying performance bottlenecks and system anomalies, thereby improving the efficiency and reliability of the systems. [76–81]

2.5 Summary

In this chapter, we have explored a wide array of tracers, agents, instrumentation, and analysis tools. Each tool has its strengths and weaknesses, which can vary depending on its adoption by the user. Consequently, it is crucial to select a tool that aligns with our specific requirements.

Low-latency systems cannot be traced in the same manner as typical cloud applications. In most cases, precision is necessary to identify the root cause of problems. Therefore, using observability agents to collect metrics might not be suitable for our use cases. Solutions like **LTTng** would be a more appropriate choice for obtaining fine-grained resolution of host metrics.

Except for host metrics, all other TD can be collected using **OTel** APIs and SDKs. The data gathered will be tool and vendor-neutral, and can be easily utilized by the vast majority of the ever-growing list of OBs.

On the other hand, **Trace Compass** will be extremely beneficial when the time comes to create custom analyses and visualizations, not supported by popular OBs. It could help answer questions like, “What was the CPU usage at time X using a 5ms resolution?” or

“What was the number of concurrent requests on our servers during a specific operation X?”.

CHAPTER 3 ENVIRONMENT SETUP

In this chapter, we outline the specific hardware and software environment wherein we applied the solution that is further detailed in the subsequent chapter article. Additionally, we provide links facilitating access to the source code of the tools that we have developed.

3.1 Hardware and software configuration

The work conducted within this research has been executed and tested using two computers : a target host, where we run the microservices that we want to trace, and a remote host, to which we can send TD. The target host and the remote host are situated within the same local network, with a distance of two switch hops separating them.

The target system is equipped with an Intel Core i7-7820X CPU, which boasts 8 physical cores, 16 virtual cores, and a base frequency of 3.60 GHz, along with 32 GB of RAM. The remote system is a virtual machine powered by a QEMU Virtual CPU, featuring 8 virtual cores, a base frequency of 3.60 GHz, and 8 GB of RAM. Both hosts run the Ubuntu 20.04 operating system.

We used LTTng v2.13, the OpenTelemetry v1.8.1 C++ client and the OTel Collector Contrib v0.80.

3.2 Developed tools

Given the absence of an official OTel C library, we crafted C bindings around the official C++ library. This enabled us to trace our applications effectively using the newly developed C bindings library. The wrapper, which includes basic examples of TD collection and LTTng exporters for every type of TD - including request traces, application metrics, and logs - can be accessed at <https://github.com/dorsal-lab/opentelemetry-c>.

To evaluate the overhead of the proposed tracing architecture, we benchmarked the overhead of collecting each type of TD in different scenarios. The benchmark source code and results are accessible at <https://github.com/dorsal-lab/opentelemetry-c-performance>.

To showcase the practical applicability of our solution through a use case. We have developed a ZeroMQ client and a proxy server application in C, which we subsequently traced using our wrapper. The application can be accessed at <https://github.com/dorsal-lab/opentelemetry-c-demo>.

We subsequently created the replayer project (see Figure 4.6), which reads TD generated by our wrapper and sends them to an OTel collector. The OTel collector, was configured to dispatch the corresponding TD to the appropriate OBs: Jaeger for request traces, Loki for logs, and VictoriaMetrics for metrics. These OBs serve as data sources in Grafana. As a result, all TD collected can be conveniently viewed using Grafana alone. The replayer project can be accessed at <https://github.com/dorsal-lab/opentelemetry-c-replayer>.

CHAPTER 4 ARTICLE 1: TRACING LOW LATENCY MICROSERVICES

Authors:

Eya-Tom Augustin Sangam

Polytechnique Montréal

`eya-tom-augustin.sangam@polymtl.ca`

Michel Dagenais

Polytechnique Montréal

`michel.dagenais@polymtl.ca`

Hadi Dewan

Ciena

`hdewan@ciena.com`

Authors contributions:

Eya-Tom Augustin Sangam: Conceptualization, Methodology, Validation, Investigation, Writing. **Michel Dagenais:** Conceptualization, Methodology, Validation, Investigation, Resources, Writing review and editing, Supervision, Project administration, Funding acquisition. **Hadi Dewan:** Conceptualization, Methodology, Validation, Investigation, Resources.

Keywords: distributed tracing, kernel tracing, software performance, software observability.

Submitted to: Software: Practice and Experience, November 21 2023.

Abstract: Microservices architectures, which enhance flexibility, boost robustness, and simplify updates by dividing applications into small autonomous services, address the complexities of monolithic architectures. Low latency microservices, designed for minimal delay, are crucial for real-time response industries. However, their distribution across various networked systems makes debugging and monitoring challenging. As a system administrator, it becomes vital to gather telemetry data (TD) from microservices to ensure seamless system monitoring, optimization, and reliability. Doing so enables the easy identification of performance bottlenecks, anomaly detection, and facilitates data-driven decision-making. This article emphasizes the collection of TD from microservices which require low latency and have limited storage and other resources. The criticality of analyzing TD across multiple hosts,

implementing filters to decrease the volume of TD saved, and mitigating impacts on system performance is discussed. In response to these challenges, we propose a tracing architecture that significantly reduces the overhead on the system being traced by an order of magnitude. This architecture involves a combination of LTTng and OpenTelemetry. The process consists of two phases. The online phase involves structuring and collecting TD with OpenTelemetry, and storing it in CTF files using LTTng. The offline phase, on the other hand, concentrates on the execution of analyses.

4.1 INTRODUCTION

Microservices, often referred to as the microservices architecture, represent an architectural approach that breaks down an application into a suite of small, autonomous services, each centered around specific business capabilities. These services, each deployable, operational, and scalable on their own, offer a high degree of flexibility and robustness. The shift towards microservices is driven by the challenges posed by monolithic architectures, which become increasingly complex to comprehend, develop, test, and deploy as applications expand. Microservices address these issues with their modular and independent nature, simplifying updates and mitigating risks.

Low latency systems refer to computing environments where minimal delay (latency) is critical to their functioning. These systems are designed to process a very high volume of data messages with minimal delay. They are used in industries and applications where real-time response or processing is required. Examples include financial services for high-frequency trading, telecommunications, and real-time gaming. Microservices began to be used in low latency systems around the mid-to-late 2010s, alongside the rise of cloud computing, containerization technologies (like Docker), and orchestration tools (like Kubernetes). Low latency microservices refer to services designed and optimized to respond to requests or process data with minimal delay.

In a typical microservices-based architecture, a single business process may involve numerous microservices, possibly running on distinct systems or containers. This dispersion adds a layer of complexity to debugging and monitoring, compared to their monolithic counterparts. Thus, tracing and monitoring has emerged as crucial elements in maintaining optimal performance and functionality in contemporary microservice architectures. This involves monitoring the health, performance, and interactions of the various software and hardware components that constitute the entire system.

When tracing low latencies microservices, we need to consider the characteristics of the hosts

running them. Often, these hosts are equipped with limited hard drive storage, necessitating the implementation of a filtering mechanism to minimize the volume of data saved on the disk. For instance, we should have the flexibility to determine at runtime if we want to save traces of heartbeat signals, particularly for distributed services that frequently transmit synchronization messages. Furthermore, many applications are run on hosts with constrained resources. The installation of any data collection agent, or database, could significantly impact the application behavior. Based on the requirements provided, the following elements of the problem can be identified:

Efficient telemetry data collection The challenge lies in efficiently gathering telemetry data (TD) while managing the limited resources of our microservices hosts.

Impact on system performance The process of data collection could itself impact the system performance. The challenge is to minimize this impact while ensuring the necessary data for monitoring and optimization is collected.

Network traffic minimization TD collection could involve significant network traffic. The problem here is to reduce the amount of data sent across the network, which could potentially affect the system performance.

Structured data collection For effective analysis, the TD must be collected and stored in a structured manner. The problem here is to ensure structured data collection that facilitates subsequent analysis.

Data analysis across multiple hosts By examining the TD collected from various hosts, we can identify performance bottlenecks. The challenge lies in suggesting techniques for this analysis.

The research detailed in this article aims to address the following two questions: (i) *Which tools and strategies are most appropriate for the efficient collection of TD from resource-limited, low-latency microservices?* (ii) *How can the TD collected from multiple hosts, be utilized to detect performance bottlenecks and system anomalies?* Specifically, the main contributions of the current paper are: (1) A survey of current state of the art tools used to trace distributed systems in general, and low latency systems in particular. (2) A new tracing architecture reducing the overhead on the system being traced.

The remainder of this paper is organized as follows. Section 4.2 provides a review of key concepts necessary for understanding the work presented, and establishes the nomenclature used in this article. Section 4.3 surveys related work. Section 4.4 presents various tracing

architectures that can be utilized. Section 4.5 benchmarks these architectures and highlights the one with the lowest overhead. Section 4.6 provides a practical use case of low latency microservices traced using the proposed architecture. Finally, Section 4.7 suggests potential areas for future research and concludes this paper.

4.2 DEFINITIONS AND NOMENCLATURE

4.2.1 Tracing

Tracing is a technique used in software debugging and performance analysis to record the sequence of operations or function calls made by a program or a system. It allows developers and system administrators to understand the behavior of a system during its execution [1]. Tracers are the tools or utilities that facilitate this process. They monitor and record the events of interest in a system. These events can range from system calls, function entry/exit, interrupts, and exceptions, to other specific application or system-level events. The study will concentrate on three distinct categories of tracers:

Kernel Tracers These tracers monitor activities at the kernel level. They can provide valuable insights into how the operating system is interacting with the hardware and managing resources. Examples of Kernel tracers include Ftrace and LTTng.

Userspace Tracers These tracers operate at the application level, tracking the behavior of user programs. They can help developers understand the execution flow of their code, identify performance bottlenecks, or debug problems. Examples include ltrace and LTTng-UST.

Distributed Tracers These are used in distributed systems or microservices architectures. They help in understanding the interactions between different services in a system, which can be crucial for debugging complex distributed issues. Examples include OpenTelemetry clients.

Userspace and kernel tracers frequently depend on tracepoints for their operations. A **tracepoint** is a statement, placed directly in the source code of the kernel or the application, that provides a hook to invoke a probe. A **probe** is a function that is called whenever the tracepoint is encountered at runtime. A probe usually performs a custom task and is either implemented by the tracer or by the user. Typically, a probe is small and fast, which keeps it from slowing down the system. An **event** marks the encounter of a tracepoint at runtime. Depending on the kind of tracing, the event can have a logical meaning, such as a context

switch, or can simply represent a location in the code, such as a function entry or exit. An event is instantaneous, generates a trace record without any duration, and is typically marked with a timestamp for accurate tracking.

4.2.2 Telemetry data

While the term “telemetry” has been in use for decades within various fields, like space exploration, the OpenTelemetry project certainly has played a significant role in popularizing it in the context of software observability and distributed systems. In this document, telemetry data (TD) refers to request traces, host and application metrics, and logs.

Request traces

Request traces, also known as distributed traces, are a crucial component for understanding the entire process when a request is made to an application. This holds true whether your application is a single-database monolith or a complex network of services. By providing a comprehensive view of the complete “path” that a request follows in your application, traces enable you to get a clear understanding of the process [2].

Spans serve as the foundational building blocks of request traces. Essentially, a span encapsulates a single operation or unit of work performed within the system. The encapsulation may include, but is not limited to, the following components:

- A specific name identifying the span.
- Parent span ID, which remains empty for root spans, indicating the hierarchical relationship.
- Start and end timestamps, providing the exact time markers for the span life cycle.
- A context, which holds the trace identifiers and options that are propagated to child spans.
- Attributes, which are key-value pairs that provide additional details regarding the span.
- Events, which are time-stamped annotations on the span to describe noteworthy moments in the span lifetime.
- Links, which provide a connection between the span and other spans, which may be in the same or different traces.

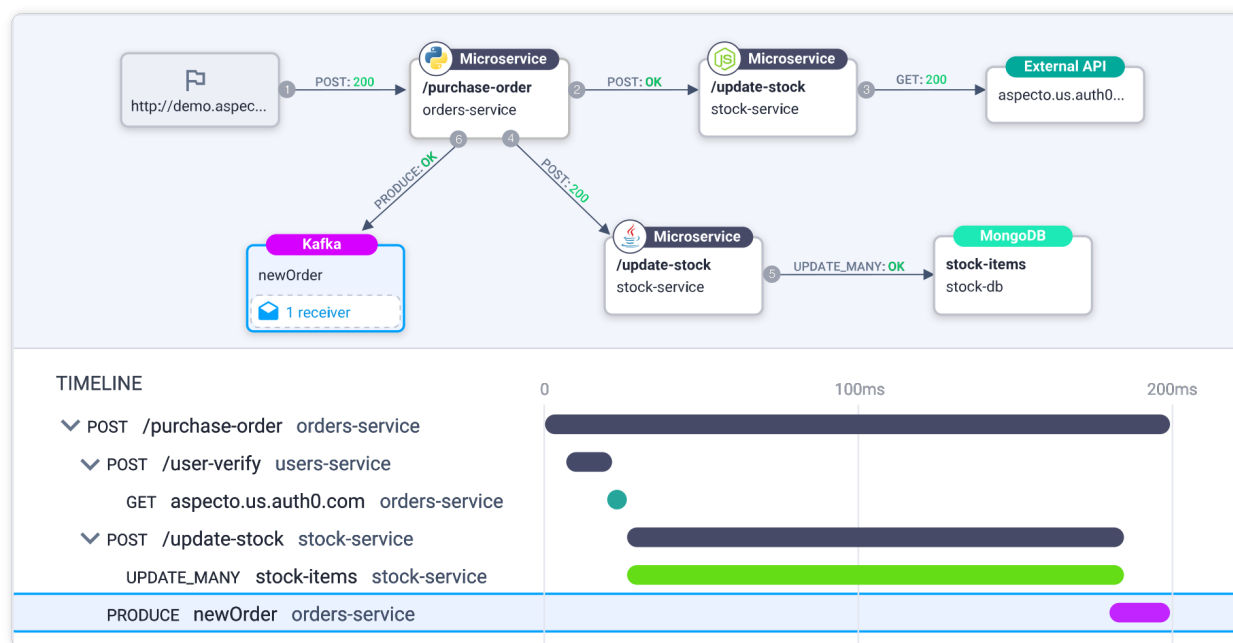


Figure 4.1 **Spans tree and swimlane visualizations.** The upper section is a span tree diagram illustrating a purchase order transaction, while the lower section shows a span swimlane diagram of the same operation. Image source [82]

- Status, an indicator of the outcome of the span - success, error, or cancellation.

Figure 4.1 provides an illustrative example of traces and spans.

Request tracing and context propagation Context propagation in request tracing refers to the process of passing along metadata from one service to another in a distributed system. This metadata, often referred to as the “context”, typically includes trace identifiers, identifiers of the individual operations, and other relevant information.

Context Propagation is the core concept that enables distributed tracing. The purpose of context propagation is to create complete, end-to-end traces of transactions as they traverse through various microservices in a distributed system. This allows for easier debugging, improved understanding of system behavior, and better performance analysis [3].

In a practical sense, context propagation usually involves adding specific headers to HTTP requests or messages shared between services. These headers are then read by the receiving service and can be used to create a child span in its tracing system, thus creating a link between the services in the trace.

Host metrics

Host metrics allow you to monitor the performance and resource allocation of the hosts and processes that your critical applications depend on. These metrics provide a fundamental understanding of how your host resources are being utilized and can help identify potential performance issues or bottlenecks in your system. Host metrics typically focus on the fundamental resources of your system. Here are some common examples [4] :

- **CPU Usage:** Refers to the percentage of CPU resources in use and can be further broken down into user time, system time, and idle time.
- **Memory Usage:** The amount of memory being used versus the total available. This can include metrics for used memory, free memory, cached memory, and memory available for applications.
- **Disk I/O:** The amount of input and output operations happening on your disk. This can help identify if your disk has become a bottleneck.
- **Network Traffic:** Data such as the rate of data transmission and reception, packet loss, and network errors can help identify network-related issues.
- **Load Average:** A measure of the workload of the CPU and I/O system. It gives the average number of jobs in the execution queue over a certain period.
- **Disk Usage:** The amount of disk space used versus total available. This can help identify if you are running low on disk space.
- **Number of Running Processes:** The count of processes currently running on the host.
- **Number of Open Files:** The count of files currently being accessed.
- **Swap Usage:** The amount of swap space used versus total available.

Application metrics

An application metric is a real-time quantification of a service performance [5]. The instant at which these quantifications are recorded is termed a metric event. This event comprises not only the measurement but also the timestamp of its capture and any relevant metadata. Metrics related to application performance and its request handling serve as vital barometers of system availability and efficiency. Other custom metrics can reveal the correlation between

availability markers and user experience or business operations. The data collected from these measurements can be used to raise alerts during service disruptions or schedule automatic deployments in response to a rise in traffic.

Here are some examples [6]:

- Request Rate: The number of requests your application is handling per unit of time. High request rates could indicate a high load or potentially even a denial-of-service attack.
- Error Rate: The number of requests resulting in an error. This could indicate issues with your application code or downstream services.
- Response Time: The time it takes for your application to respond to a request. High response times could indicate performance issues.
- Throughput: The amount of data your application is processing in a given amount of time.
- Garbage Collection Metrics: If your application uses a garbage-collected language like Java or Go, metrics like the frequency and duration of garbage collections can be important indicators of performance.
- Database Query Times: The time taken to execute database queries. Long query times could indicate database performance issues or inefficient queries.
- Cache Hit/Miss Rates: If your application uses caching, the cache hit and miss rates can tell you how effectively your cache is being utilized.
- Queue Depth: If your application uses queues, the depth of the queue (number of messages waiting to be processed) can indicate if your application is able to keep up with the load.

Application logs

A log is a timestamped record, either structured (recommended) or unstructured (more ambiguous free form text), with metadata. Most programming languages have built-in logging capabilities, or well-known widely used logging libraries. While logs can function as a standalone data source, they can also be appended to spans. Logs often contain detailed debugging/diagnostic info, such as inputs to an operation, the result of the operation, and any supporting metadata for that operation [7].

4.2.3 Observability backends

Observability lets us understand a system from the outside, by letting us ask questions about that system without knowing its inner workings. Furthermore, it allows us to easily troubleshoot and handle novel problems (i.e. “unknown unknowns”), and helps us answer the question “Why is this happening”. Observability heavily relies on TD [8].

An Observability Backend (OB) is a platform that collects, stores, analyzes, and helps visualize TD. The main goal of an OB is to provide a comprehensive view of your system state and behavior over time. This helps in performance optimization, troubleshooting, and understanding the overall health of your system. OBs provide features (not necessarily all) for querying and visualizing TD, setting alerts based on specific conditions, and often include support for advanced analytics and machine learning, to detect anomalies and predict system behavior.

OBs can be categorized into three main types: metrics backends or time series databases, log backends or column stores, and request trace backends or request trace databases. Metric backends focus on storing and analyzing numerical data that measures system or application attributes over time, aiding in real-time monitoring, alerting, and performance trend analysis. Log backends handle the storage and analysis of log entries, providing detailed contextual information about events. Lastly, request trace backends focus on the storage and analysis of traces, which are records of a request path through the system.

Each type of OB is designed to handle a specific type of observability data. However, as shown in this recent study [9], many modern observability services combine all three types to provide a more comprehensive view of the system behavior.

4.3 RELATED WORK

4.3.1 Distributed tracing

Distributed tracing, as a concept, began to gain prominence with the rise of microservices architecture. As systems became more complex and distributed, it became increasingly difficult to understand the flow of requests and data across different services. Traditional monitoring and debugging tools were not efficient for these distributed systems. The concept of distributed tracing was then introduced to tackle these challenges.

The concept of distributed tracing was popularized by Dapper [11], a system developed by Google. Introduced in a 2010 paper, Dapper was one of the first large-scale distributed tracing systems. Dapper is part of a long tradition of tools and research aimed at solving

the challenges of understanding behavior in distributed systems. Before Dapper, there were several other tools and research projects aimed at this issue. Pinpoint [12], developed by teams at the University of California, Berkeley, and Stanford University, along with systems like Magpie [13], developed by Microsoft Research, and X-Trace [14], from the AMP Lab at UC Berkeley, were early attempts to provide detailed tracing for distributed systems. Each of these tools represented significant advancements in the field, contributing to the evolution of distributed tracing technologies and methodologies.

Dapper inspired many open-source distributed tracing systems. These include Zipkin [16], created by Twitter, and Jaeger [21], developed by Uber Technologies. As more and more distributed tracing systems began to emerge, the need for a standard way to collect, manage, and analyze traces became apparent. This led to the creation of OpenTracing [17] and OpenCensus [26], two open-source projects that aimed to provide standardized APIs for distributed tracing and metrics.

In 2019, these two projects merged to form OpenTelemetry (OTel), under the umbrella of the Cloud Native Computing Foundation (CNCF) [27]. OTel continued to evolve and became today an open-source observability framework that generates, collects, and manages TD from software services. Crucially, OTel is vendor- and tool-agnostic, meaning that it can be used with a broad variety of OBs, including open-source tools, as well as commercial offerings.

OTel consists of the following major components:

OTel specification The specification [29] defines standard procedures and conventions for collecting, processing, and managing TD. This specification serves as a guide for all APIs, SDKs, and protocol implementations.

The specification outlines the key concepts and terminologies used in OTel and provides a comprehensive framework for capturing and managing TD. It defines the structure and format of traces and metrics, as well as the protocols for sending them to other systems.

It also outlines standards for context propagation, which allows for the correlation of telemetry across multiple distributed systems. Other conventions covered by the specification include resource identification, semantic conventions for common scenarios, and error handling.

The OTel specification is continually updated and refined by a diverse community of contributors to reflect the evolving needs of developers and the broader technology landscape. Its goal is to provide a consistent and unified approach to observability across all programming languages and platforms.

OTel protocol (OTLP) OTLP [30] is a general-purpose TD delivery protocol designed in the scope of the OTel project. Its specification describes the encoding, transport, and delivery mechanism of TD between telemetry sources, intermediate nodes (such as collectors) and OBs. The OTel Protocol (OTLP) operates on a request-response model. The client sends a request to the server and waits for a response. The client can be any application or service that produces TD, such as traces, metrics, and logs. Typically, clients are web servers, microservices, or any instrumented software that generates TD. The server, on the other hand, would be the observability backend or any platform that receives, processes, and possibly visualizes this TD. This could be a service like Prometheus, Jaeger, or any application that is built to handle the ingestion and analysis of TD.

OTel Protocol (OTLP) uses Protocol Buffers (protobuf) as its data format and gRPC/HTTP for data transmission, providing advantages like efficient serialization and strong type checking. Protobuf provides language-neutral and platform-neutral schemas, allowing developers to easily define how data should be structured. This universal language support is advantageous, because it enables OTLP to be easily implemented across various languages. Developers can leverage protobuf generated APIs, which are available in numerous languages, to serialize the structured data (TD in this case) to protobuf format, and deserialize from the protobuf format to the data structures in their programming language. This facilitates the transmission of TD using OTLP.

Semantic conventions Semantic Conventions [31] are agreed-upon standards for how to record specific types of data, like HTTP requests, database calls, or container information. They define a common set of attributes, their type, and their meaning to be used when recording TD. This ensures consistency and uniformity of data across different services, languages, and backend observability platforms, making the data more useful and interpretable.

APIs and SDKs APIs [32–37] define how to generate TD. These APIs are language-specific and provided through Software Development Kits (SDKs) for various popular languages like Java, Python, JavaScript, Go, etc. Key OTel APIs include the Tracing API for distributed request trace collection, the Metrics API for numerical data capture, the Baggage API for context propagation across process boundaries, the Context API for context management, and the Resource API for describing the running compute unit. While APIs are used to instrument the code and generate TD, the SDKs [38–41] are responsible for processing and exporting that data. SDKs are the implementations of the APIs. They are language-specific and provide the tools needed to capture, process,

and export TD from your services to the backend observability platforms. The SDKs include components for tracing and metrics, and handle tasks such as span processing, metrics aggregation, and data exporting. They also provide additional features like context propagation, batching, retry mechanisms, and configurable sampling strategies.

Library ecosystem and automatic instrumentation The library ecosystem offers a wide range of language-specific libraries that provide automatic instrumentation for many popular frameworks and libraries [42]. These libraries handle the task of injecting the necessary telemetry collection code into your application, reducing the manual effort needed to instrument your code. These instrumentations capture relevant traces, metrics, and context automatically, when services using these libraries or frameworks are run, making it easier to adopt OTel in projects.

Collector The Collector [43] is a component that can receive, process, and export TD. It is designed to be a vendor-agnostic implementation and can be used as an agent (running alongside the application) or as a standalone service. Initially, the collector receives TD from applications using multiple protocols, including OTLP, Jaeger, Zipkin, and others. Next, it processes the received data, which can involve transforming, aggregating, or enhancing the data with additional metadata, or even reducing it through sampling. Finally, it exports the processed data to one or more observability backends, once again supporting multiple protocols. Moreover, the Collector supports extensions, which are not part of the data path but provide supporting capabilities like health checks and diagnostics.

4.3.2 Low latency software tracing

Low latency software tracing is focused on tracking the operations of the kernel or a software program with minimal overhead. This type of tracing provides detailed visibility into the kernel or flow of the application, making it easier to identify and diagnose performance issues. To achieve low latency, tracing tools are designed to have a minimal overhead, ensuring that the tracing process does not significantly change the whole system behavior.

LTtng [48], a tracer for both kernel and user-space in Linux, is engineered with a prime focus on performance. It has demonstrated a minimal impact of just 2% on typical high-intensity workloads, making it the most performant kernel tracer for Linux, as reported in reference [46]. This performance efficiency sets LTtng apart from other tracers such as SystemTap [57], ftrace [44], and eBPF [56], which exhibit significant latency increases when used on multicore systems [49].

One of the most powerful features of LTTng is its fine-grained control over tracing activities. Users can precisely specify what events to trace, thereby reducing system overhead and enhancing performance. This control is implemented through tracing sessions, channels, and events. Users can create a tracing session, define channels within that session, and configure each channel with various parameters such as buffer size. Multiple events can be associated with each channel and users can enable or disable these events individually or in groups, offering meticulous control over data collection. Moreover, LTTng supports both inclusion and exclusion filters for events. This allows users to determine conditions under which an event should be traced, further refining control over tracing activities. This level of control allows users to concentrate on specific areas of interest in their system, decreasing the collection of irrelevant data and making trace data analysis more manageable and efficient. LTTng also saves trace data to disk or sends it over the network for subsequent offline analysis, relying on the Common Trace Format [51], a standardized format developed for high trace event throughput. It also offers a “flight recorder” mode, further reducing overhead by only flushing buffered events to disk upon a user snapshot request [50].

4.3.3 Other tools and techniques

Observability agents

Observability agents are software components installed on a host or within an application that collect and send TD to a centralized monitoring system. Examples include the OTel Collector [43], Prometheus Node Exporter, Fluentd, Fluent Bit, and Elastic Beats. Additionally, many cloud-based observability services offer their own specialized agents, such as The New Relic Agents, AWS CloudWatch Agent, Google Cloud Operations Agent, and Azure Monitor Agent.

All these agents, despite their differences, share a common design structure. They collect various types of data depending on their design and purpose, process the data into a format comprehensible by the central monitoring system, and then transmit the processed data to the central system. For system metrics, this could involve reading from specific files in the `/proc` or `/sys` filesystems on Linux, using system APIs, or executing system commands and parsing the output. For logs, this can involve reading log files, listening on a network port, or subscribing to a messaging system. They are designed with scalability and efficiency in mind, to handle the monitoring needs of large distributed systems, without consuming significant system resources.

Each agent can be configured as per the specific needs of the system it is monitoring. They

are designed to be compatible with a wide range of systems and platforms, and to handle failures gracefully, often buffering data in case of network issues or downtime of the central system. Furthermore, these agents are designed with security considerations, to ensure secure data transmission and access controls.

Trace Compass

Trace Compass [72] is an open-source application used to visualize and analyze traces in Linux systems. It provides views, graphs, metrics, etc. to help extract useful information from traces, in a way that would be otherwise too hard to process by a human or would take a long time.

Real-time information about a Linux system can be obtained by periodically recording metrics from files located in the `/proc` directory. Tools like `top` and `ps` use this interface, parse the content, and format it for display. This technique provides system statistics at a sampling frequency that is determined by the interface. In contrast, kernel tracing does log all events in a time-ordered manner. Rather than aggregating metric values at pre-defined intervals, kernel tracing allows for the recovery of these values, at any desired resolution, afterward from the trace. This offers a more granular and accurate view of system behavior over time.

Depending on the workload of the targeted system, and the number of active tracepoints, a tracer such as LTTng can generate traces of several hundred megabytes of data in just a few seconds of execution. This vast amount of data cannot be manually analyzed. Furthermore, trace events reveal the low-level behavior of the kernel, which requires a deep understanding of kernel internals for analysis. In many instances, the meaning of an event may depend on previous events. Therefore, to extract valuable information from a kernel trace, fast and reliable analysis tools are necessary. The precise role of trace analysis tools is to scan the traces based on defined algorithms, and present a synthesis or a more useful view to the user.

A good example of a trace analysis algorithm is the calculation - from only kernel trace events - of higher-level metrics, such as CPU usage rate. Giraldeau et al. [73] proposed computing these metrics, with an arbitrary precision chosen during the analysis, from the kernel traces collected on the targeted system. In the case of CPU usage, it suffices to consider only the events of the trace that correspond to a context switch. These events contain the identifier of the process that becomes active, as well as the identifier of the CPU core on which it is scheduled. Thus, for each processor, it is possible to calculate all the intervals that correspond to the execution of a process, and from there the CPU usage rate over a given interval. Modern trace analysis tools like Trace Compass include this type of analysis for kernel traces, and offer an interactive view that the user can browse to identify anomalies

present during execution. For example, it is possible to pinpoint at a glance the moment in the trace corresponding to a particularly high total memory allocation to find the root cause. Some analyses also allow detailing the state changes of a thread over time, as shown in Figure 4.2.

Combining distributed and kernel tracing

Userspace events can help provide access to application activity unknown to the kernel. Gelle L. [74, 83] suggests combining distributed tracing with low-level software tracing to identify the root cause of spans latencies. The solution involves a hybrid trace collection to capture and synchronize both kernel and distributed request events using LTTng and LTTng-UST. The distributed tracer is instrumented to add threading information directly into spans, to facilitate postmortem synchronization. As a result, both kernel and distributed traces can be opened in a Trace Compass experiment, and thread states can be marked directly on spans.

4.3.4 Summary

We have explored a wide variety of tracers, agents, instrumentation, and analysis tools. Each tool has its strengths and weaknesses, which can vary depending on its adoption by the user. Consequently, it is crucial to select a tool that aligns with our specific requirements.

Low-latency systems cannot be traced in the same manner as typical cloud applications. In many cases, precision is crucial to pinpoint the root causes of problems. Consequently, using observability agents to collect metrics is often not suitable for our use cases. Tools like LTTng could serve as a more appropriate choice for obtaining fine-grained resolution of host metrics. However, using LTTng to collect other types of telemetry data may not be straightforward, as LTTng can be perceived as just a sophisticated logging library. For instance, for request traces, mechanisms for trace context propagation need to be defined on top of LTTng - mechanisms which are already fully handled by modern distributed tracers like OTel. Moreover, data collected by LTTng cannot be used directly by existing OBs. At the analysis stage, this data must be converted to standard OTel schemas, that all analysis tools can process.

Except for host metrics, all other TD can be collected using OTel APIs and SDKs. The data gathered will be tool and vendor-neutral and can be easily utilized by the vast majority of the ever-growing list of OBs. However, OTel does not offer as much fine-grained control over tracing sessions as LTTng does. Using OTel APIs/SDKs as a way to instrument our microservice, and logging TD generated to LTTng, allows us to remove both tools limitations

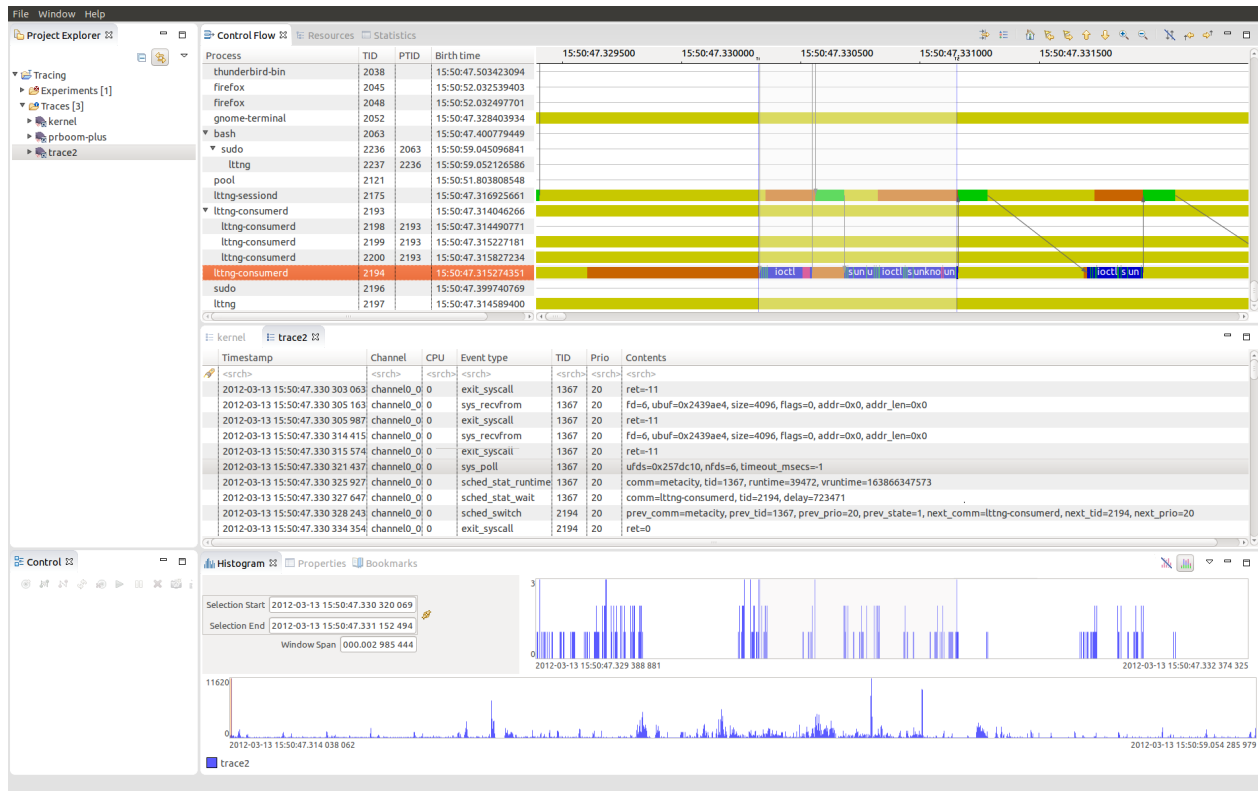


Figure 4.2 **Trace Compass Control Flow View**. Visual Representation of the Progression of Individual Execution Threads. Each state is symbolized by a distinct color. Yellow indicates a blocked execution thread, Orange for a thread awaiting CPU, Green for a thread in active execution, and Blue for the execution of a system call.

(cf. Section 4.4).

On the other hand, Trace Compass will be extremely beneficial when the time comes to create custom analyses and visualizations, not supported by popular OBs. It could help answer questions like, “What was the CPU usage at time X using a 5ms resolution?” or “What was the number of concurrent requests on our servers during a specific operation X?”.

4.4 COMBINING OPENTELEMETRY AND LTTNG

As a reminder, we want to collect host metrics, application requests traces, metrics and logs. As outlined in the related work summary, host metrics could be derived directly from kernel events with high precision, and the combination of LTTng and OTel effectively mitigates the limitations of both tools when collecting application TD. Figure 4.3 demonstrates the application TD flow graph. There are two high-level options to consider: instrument our application using the OTel API/SDKs, or manage the instrumentation ourselves.

4.4.1 Using OTel instrumentation

OTel application instrumentation refers to the process of integrating OTel APIs and SDKs into an application to collect application TD. The OTel project provides standardized tools, methods, and definitions to make this process more streamlined across various platforms and languages.

Collecting request traces

The OTel Tracing API [34] and SDK [41] provide methods to create and end spans. When you create a span, you define its operation name, start time, and other optional attributes. When the operation is complete, you end the span, and it is ready to be exported. The API also handles the propagation of the trace context across process boundaries. This means that when an operation triggers another operation in a different process, the trace context is carried over, allowing OTel to link spans together across microservices or other distributed components. Spans can be enriched with attributes (key-value pairs that provide additional information about the span), events (specific points in time during a span), and links (references to related spans). The Tracing SDK supports sampling which is a mechanism to control the noise and overhead introduced by OTel, by reducing the number of traces samples collected and sent to the backend. Finally, once the span data is collected, it can be sent to various backends by using exporters (cf. Exporting collected application TD). OTel provides several exporters

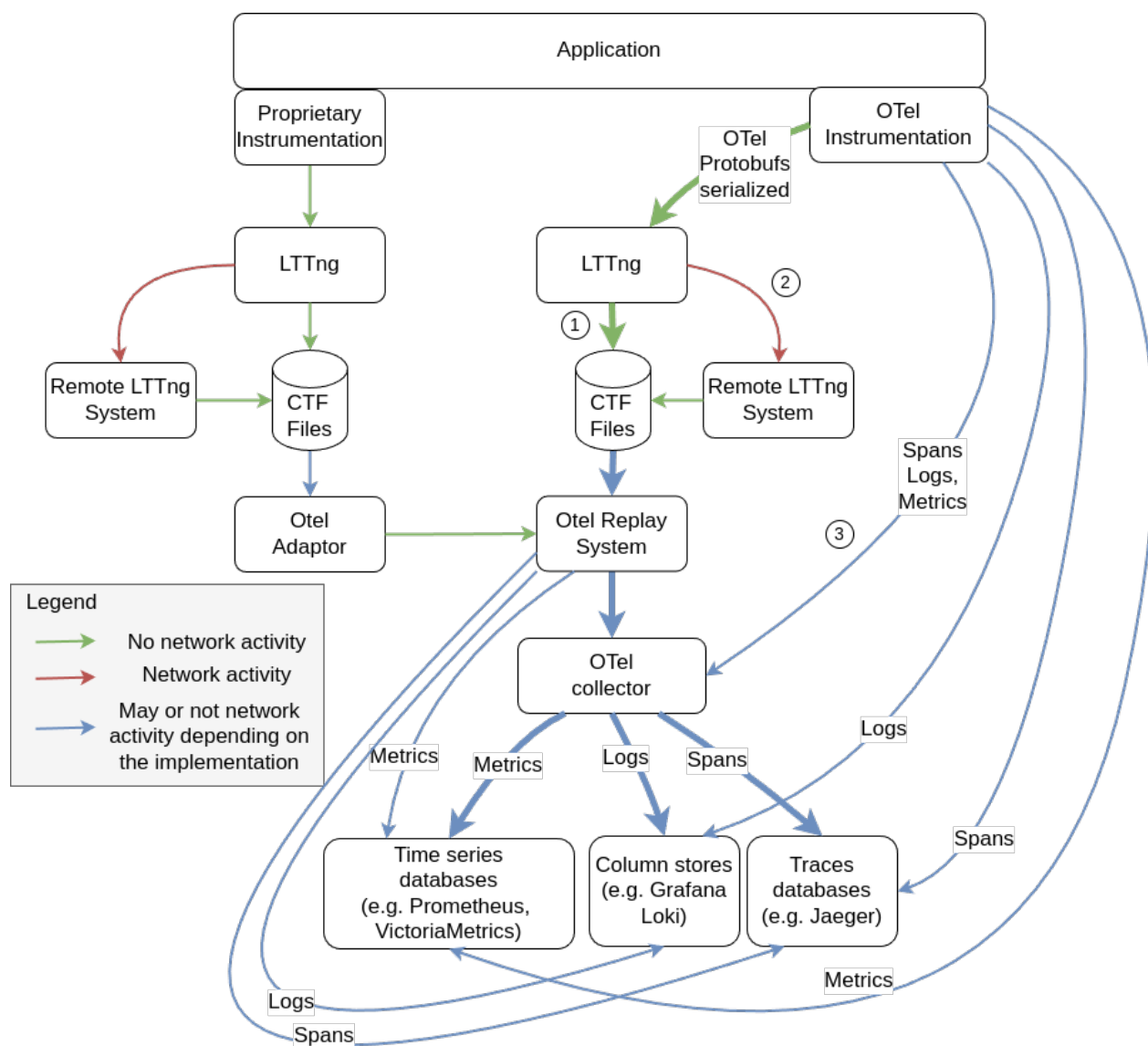


Figure 4.3 Different ways of combining OTEL and LTTng to collect application telemetry data. The application host runs LTTng and is instrumented using OpenTelemetry or proprietary instrumentation. All other components could be run either on the application host or an analysis host. Arrow color codes provide indications of network traffic. Every time network traffic is involved, telemetry data move from one host to another.

out of the box (e.g. Jaeger Exporter, Prometheus Exporter), and you can also create your own.

Collecting application metrics

The OTel Metrics API [35] and SDK [40] offer methods to create a variety of instruments to represent different metrics types, including Counters, Asynchronous Counters, Histograms, Gauges, Asynchronous Gauges, UpDownCounters, and Asynchronous UpDownCounters.

Synchronous instruments like Counters, Histograms, and Gauges capture metrics in response to specific events or operations in your code. On the other hand, asynchronous instruments collect metrics at predefined intervals, independent of particular events or operations. The choice between these depends on the metric nature and the potential impact on system performance. For some metrics, capturing data for every event, with synchronous instruments, may be too resource-intensive, making asynchronous instruments a more suitable choice.

Instruments such as Counters, which are cumulative and represent a monotonically increasing value, can only increase or reset to zero at startup. They are ideal for tracking the number of processed requests, completed tasks, or encountered errors. UpDownCounters, similar to Counters, but with the ability to increase or decrease, are useful for metrics that fluctuate, like the number of active system users or the current queue size. Histograms, collecting observed values into predefined buckets, are great for recording value distributions. For instance, you might use a Histogram to document the distribution of request sizes, giving you insight into size variance, not just the average. Gauges report the last recorded value at collection time. All these OTel instruments are tasked with reporting measurements — metric data points that encapsulate a value and optional key-value attributes.

Collecting application logs

OTel offers two APIs to manage application logs: the Event API [37] and the Log Bridge API [36]. It is important to note that application developers are generally discouraged from directly using the Log Bridge API. This API is primarily intended for library authors to construct log appenders, which utilize the API to establish a bridge between existing logging libraries (like log4j) and the OpenTelemetry log data model. On the other hand, the Event API is more developer-centric. Both application developers and instrumentation authors are encouraged to directly use the Event API. The Event API and the bridge API respectively generate LogRecords and Events. An Otel Event is a combination of event name and a LogRecord representing the event.

LogRecord contain the timestamp indicating when the log event occurred, along with severity text and a severity number to describe the importance of the log event. It also includes the name of the logger that generated the log event and the actual log message (body). Information about the originating entity, or resource, that produced the log, would be part of a LogRecord as well. Additionally, to correlate with tracing data, the TraceId and SpanId could be included. Furthermore, LogRecords would also encompass additional contextual key-value pairs, or attributes, related to the log event.

Exporting collected application TD

Exporters in OTel are components responsible for sending TD to OBs, OTel Collector or any other third party service. OTel provides several exporters out of the box to support most popular OBs. For instance, the Jaeger Exporter is designed to send trace data to Jaeger, a distributed tracing system, while the Prometheus Exporter is tailored to deliver metrics data to Prometheus, a monitoring solution and time series database. The Zipkin Exporter, similar to the Jaeger Exporter, sends trace data, but to the Zipkin distributed tracing system. In addition, there is the OTLP Exporters, which is capable of sending both trace and metric data to any backend that supports the OTLP. For debugging purposes, the Console Exporter is available to print trace or metric data to the console.

OTLP exporters function by formatting and sending telemetry data in the OTLP format to any backend that supports it. The process initiates with data conversion, where trace and metric data are transformed into the OTLP data format. This step includes mapping spans, metrics, and eventually logs into the corresponding OTLP Protobuf data structures. Following conversion, the data undergoes serialization, typically into Protobuf. This language-neutral, platform-neutral mechanism provides an extensible way to serialize structured data. Once serialized, the data is primed for transmission. The OTLP exporter dispatches the serialized data to the configured backend endpoint using gRPC or HTTP/protobuf. The choice of transport protocol, be it gRPC or HTTP/protobuf, can typically be configured in the OTLP exporter settings.

Drawing inspiration from OTLP exporters, we have developed LTTng exporters. These exporters carry out the same data conversion and serialization processes. However, instead of transmitting serialized Protobuf data, LTTng exporters log this information directly into LTTng. Additional fields (e.g. service name) can be appended to the logged event containing the serialized Protobufs, serving as filtering fields at runtime. All TD collected will be logged into CTF files located in the target filesystem, or on the remote filesystem if the LTTng relay daemon is used.

Reading TD CTF files

Once TD CTF files are created and we are prepared to visualize or analyze the collected TD, these CTF files can be transferred from the target system to the analysis host. TD CTF files can be directly visualized in tools like Trace Compass, or it can be read, deserialized, and sent to an OTel collector using OTLP. The collector is responsible for dispatching TD to the appropriate backends. This includes sending logs to column stores, metrics to time series databases, and request traces to trace databases. The component tasked with reading the CTF files and sending them to the collector is named OTel Replayer in Figure 4.3. The replayer leverages the Babeltrace library API [84] to read the binary CTF files.

4.4.2 Using custom instrumentation

Utilizing OTel instrumentation may not always be feasible. In some large-scale projects, proprietary instrumentation libraries have been established and are already deployed in production throughout all microservices. It could potentially be expensive to completely transition from the currently used instrumentation strategy to an OTel-based one. However, TD generated can be converted to conform to OTel schemas, thereby facilitating integration with OTel-compatible OBs. In this section, we will provide an example of custom instrumentation and the journey of its generated TD to OTel-compatible OBs. We use in this example LTTng as user space tracer.

Collecting request traces

The easiest way to collect spans is to log two events for each operation: one event at the start and the other one at the end of the operation. The problem with this simple approach is that spans causal relationship, deduced with request context propagation introduced by modern distributed tracers, will not be taken into account. Making spans quite useless during the analysis.

To utilize context propagation, every span must have a unique ID. When initiating a span, both its unique ID and the ID of its parent span, if it exists, must be logged. Ending a span only requires the logging of the unique ID of that particular span. When using HTTP requests, the parent span ID can be incorporated into the HTTP headers. Alternatively, it can be included in the request body or be a distinct message.

While handling a request, we first extract the parent span ID from the request. Then, generate a new unique span ID and log a “start_span” event that references both the new span ID and the parent span ID. After this, the request can be processed. After the request

has been processed, we log an “end_span” event, including only the unique ID of the span that has just finished. If handling the request involves sending additional requests, we ensure to include the current span ID in these requests to maintain the traceability of the spans. This process is summarized in Algorithm 1.

Span ID generation should be conducted in a manner that minimizes the likelihood of collisions. A prime example of an ID generator is Twitter Snowflake [85], which generates approximately sorted 64-bit IDs in an uncoordinated fashion. It achieves this by combining the current timestamp, worker number, and a thread local sequence number.

Algorithm 1 Trace Handling Procedure

```

1: procedure HANDLEREQUEST(request)
2:   parentSpanId ← getParentSpanId(request)           ▷ Extract the parent span ID
3:   spanId ← generateUniqueSpanId()                 ▷ Generate a unique span ID
4:   logEvent('start_span', parentSpanId, spanId)     ▷ Log the start span event
5:   ProcessRequest(request, spanId)                 ▷ Process the request
6:   logEvent('end_span', spanId)                   ▷ Log the end span event
7: end procedure
8: procedure PROCESSREQUEST(request, parentSpanId)
9:   if requestNeedsAnotherRequest(request) then     ▷ If the request needs another
    request
10:    newRequest ← createNewRequest(parentSpanId)    ▷ Create a new request
11:    sendRequest(newRequest)                       ▷ Send the new request
12:  end if
13: end procedure

```

Collecting application metrics

Collecting metrics is a relatively simple process. For synchronous metrics, each metric update corresponds to a logged event. Asynchronous metrics, on the other hand, require periodic event logging. To achieve this, we can utilize a separate thread that ensures events are logged at regular intervals.

Collecting application logs

We can conveniently leverage the logging capabilities provided by LTTng to efficiently capture and manage application logs.

Transforming TD collected into OTel standard schemas

All TD collected will be logged into CTF files. These CTF files can be directly visualized using tools like Trace Compass or read using Babeltrace. However, they cannot be sent directly to an OTel collector using OTLP. Instead, we need to create OTel protobufs based on the collected data.

To accomplish this, we can reconstruct a complete Span Tree using the “start_span” and “end_span” events. The root of the tree will be the span with no parent Span ID, and parent Span IDs can be used to establish parent-child span relationships. OTel requires all spans to have a 16-byte trace ID and an 8-byte Span ID. We can obtain the OTel trace ID by hashing the root non-OTel span ID, which will maintain a consistent value for the same span tree. The OTel Span ID can be directly generated by hashing the non-OTel span id. All other required protobufs fields such as the operation name, start and end timestamp will be extracted directly from span events. A similar operation will be done with OTel metrics and OTel logs protobufs. The entity responsible of such transforms is named “OTel replayer” in Figure 4.3.

Once TD are transformed into protobufs, these can be transmitted to the OTel collector. From there, the collector can distribute them to the appropriate OBs.

4.5 EVALUATION

When evaluating any tracing solution, it is essential to thoroughly measure the overhead imposed on the original system. Excessive overhead can not only mask the sources of existing latencies but also generate new ones. To assess the performance impact on individual systems, and to understand their behavior under different loads and configurations, we have carried out this crucial step.

The previous section outlined various ways of collecting and transferring TD. To assess the pathway that minimizes the overhead on the traced system, we benchmarked the overhead of all potential TD paths, from their inception using OTel to their arrival at OBs. This evaluation aims to identify the most efficient route for TD, thereby reducing the impact on the system being traced.

Table 4.1 outlines various configurations used for our benchmarks. The option codes help figuring out the TD path. For instance, when measuring the overhead of collecting request traces, the “LSRU-OTel-SP-LE” configuration group corresponds to a setup in which the application is instrumented using OTel, and request spans are individually exported locally to LTTng, with no events being discarded. This configuration group corresponds to the path

with bold arrows in Figure 4.3 and is marked as ①. Benchmarking for TD paths resulting from proprietary instrumentation has not been carried out. The primary reason for this omission is the complexity involved in achieving a fair comparison with OTel instrumentation. It would necessitate us to virtually recreate all the OTel APIs and SDKs, which is a significant undertaking. We also intentionally avoided benchmarking a direct export of TD to OBs. This approach is less applicable to production systems. As systems expand, managing direct connections from each service to the backends becomes complicated. Furthermore, switching backends would require updating and redeploying all services, which can be a challenging task.

For all tests, the target system is equipped with an Intel Core i7-7820X CPU, which boasts 8 physical cores, 16 virtual cores, and a base frequency of 3.60 GHz, along with 32 GB of RAM. The remote system is a virtual machine powered by a QEMU Virtual CPU, 8 virtual cores, a base frequency of 3.60 GHz, and 8 GB of RAM. We use LTTng v2.13 and the OpenTelemetry v1.8.1 C++ client. The application host and the remote host are situated within the same local network, with a distance of two switch hops separating them.

4.5.1 Overhead of collecting request traces

Driven by the guidelines set in the Performance Benchmark of OpenTelemetry API [86], we evaluate the overhead associated with request trace collection. Our testing methodology is straightforward: we initiate a span, perform minimal operations, and subsequently terminate the span. We have benchmarked the following scenarios to provide a comprehensive understanding of the system performance:

1. Simple (no operation): Start a span and end it right away.
2. Extract context: Start a span, extract the span context that can be sent to a remote process and end the span.
3. Adding an event: Start a span, add one span event and end the span.
4. Set a span attribute: Start a span and set a span attribute, and end the span.
5. Start span from context: Start a span from a span context created by a remote process. End the span.
6. Create a nested span: Start a span. Start a nested span inside that span. End both. Measure the time to create and end the inner span.

Table 4.1 Benchmark configuration options

Types of configuration	Option	Code	Comments
LTTng	No LTTng session running	NLS	LTTng is disabled.
	LTTng session without recording	LSWR	LTTng session is active, tracepoints are defined but there is no active recording rule. All TD are discarded.
	LTTng session recording userspace TD	LSRU	LTTng session is active, tracepoints are defined and all userspace tracepoints are active.
	LTTng remote session recording userspace TD	LRSRU	Same as above but we sending all events to the remote host using the LTTng relay-daemon.
Instrumentation	No instrumentation	NI	The application is left as is.
	OpenTelemetry	OTel	OpenTelemetry instrumentation is used.
Span Processor (only for request traces)	Simple	SP	Spans are exported directly after completion.
	Batching	BP	Spans are exported in batch of 512 spans every 5s by a separate thread.
Exporter	LTTng Exporter	LE	TD are logged to LTTng.
	Local OTel collector	LOC	TD are sent to an OTel Collector running on the target host.
	Remote OTel collector	ROC	TD are sent to an OTel Collector running on a remote host.

Scenario 1 serves as a reference. Comparing all other scenarios to it helps us understand the overhead of conducting a specific span operation. For example, the differences between scenarios 1 and 3 can aid in understanding the overhead associated with adding an event to a span.

Results of the benchmarks for all scenarios are shown in Table 4.2. “real”, “user” and “sys” statistics are respectively total time spent by all operations from start to finish (wall-clock time), the total time used directly by the process for executing instructions (CPU time) and the total time used by the kernel to execute system calls (system CPU time). When benchmarking a multithreaded system, it is crucial to consider both wall-clock time and CPU time. Operations, such as exporting spans, can be executed on a background thread. This concurrent operation can decrease the wall-clock time, as it enables the CPU to process another task simultaneously. However, this does not necessarily lessen the CPU time. Thus, wall-clock time and CPU time can show significant discrepancies.

Comparison: NLS-OTel-SP-ROC ③ vs LSRU-OTel-SP-LE ① vs LRSRU-OTel-SP-LE ② We compare three configuration groups: the first exports spans individually (Simple processor) to a remote OTel collector as they are created. The second saves spans on the disk using a local LTTng session, while the third sends spans individually to a remote LTTng session.

In the first setup, the average wall-clock time recorded to export a span is 2946 μ s for ③ NLS-

Table 4.2 Requests trace collection benchmarks

NLS-OTel-SP-LOC ③ (5 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	1,014,362	955,341	1,035,142	842,327	930,969	718,063
mean (ns)	1,975,102	1,984,238	2,005,028	1,945,079	1,986,050	1,674,941
max (ns)	16,427,715	13,078,189	14,378,366	14,739,290	4,246,470	12,380,256
median (ns)	2,168,330	2,235,832	2,000,499	2,007,091	2,257,778	1,695,893
std (ns)	376,562	351,957	373,002	405,747	333,704	387,726
real (ms)	60,475	60,529	60,632	60,318	60,550	66,213
user (ms)	7,192	7,187	7,372	7,152	7,370	11,181
sys (ms)	458	494	397	395	345	779
NLS-OTel-BP-LOC ③ (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	19,074	19,327	21,548	17,564	17,946	11,207
mean (ns)	115,143	122,008	123,043	129,831	126,346	46,075
max (ns)	647,056	434,903	477,539	440,165	293,792	252,588
median (ns)	114,738	119,487	120,451	141,345	130,353	25,066
std (ns)	12,709	13,115	13,067	12,903	12,583	7,970
real (ms)	204,822	204,957	204,993	205,108	205,052	205,489
user (ms)	3,513	3,707	3,729	3,826	3,780	4,542
sys (ms)	358	295	304	307	310	391
NLS-OTel-SP-ROC ③ (5 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	1,931,562	1,637,761	1,899,313	1,928,850	1,803,930	1,783,299
mean (ns)	2,945,936	2,936,436	2,923,788	2,908,457	2,930,099	2,767,380
max (ns)	15,251,232	16,160,800	17,865,621	19,950,437	35,345,712	14,521,650
median (ns)	2,796,951	2,912,457	2,787,925	2,551,865	2,266,497	2,771,626
std (ns)	478,621	475,818	547,792	475,268	723,732	502,058
real (ms)	65,391	65,341	65,270	65,201	65,324	77,421
user (ms)	8,079	7,940	7,870	7,710	7,970	13,724
sys (ms)	369	489	484	757	545	752
NLS-OTel-BP-ROC ③ (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	21,101	37,600	24,412	30,890	23,516	14,063
mean (ns)	116,657	123,399	124,337	129,062	128,142	45,412
max (ns)	455,129	406,377	459,674	486,441	283,063	175,988
median (ns)	117,134	121,845	124,141	128,672	125,311	44,857
std (ns)	9,668	9,600	9,865	10,308	9,873	6,815
real (ms)	204,911	205,032	205,077	205,131	205,099	205,554
user (ms)	3,663	3,749	3,789	3,888	3,883	4,737
sys (ms)	330	305	348	350	317	382

NLS-OTel-SP-LE ① (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	97,795	107,655	114,352	103,234	89,947	56,431
mean (ns)	291,046	293,832	298,934	297,913	295,996	215,863
max (ns)	909,617	889,669	942,659	790,711	605,800	522,061
median (ns)	308,049	297,158	289,732	298,188	291,719	212,653
std (ns)	22,677	21,897	22,116	23,665	21,996	21,474
real (ms)	208,426	208,504	208,608	208,584	208,540	211,074
user (ms)	6,033	6,108	6,299	6,222	6,221	8,758
sys (ms)	440	436	343	403	357	353
NLS-OTel-BP-LE ① (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	12,697	39,904	31,402	24,377	22,494	12,482
mean (ns)	115,924	124,190	125,949	132,926	128,032	46,306
max (ns)	406,575	384,695	834,295	565,522	320,963	216,068
median (ns)	117,818	119,784	123,243	133,417	130,438	96,738
std (ns)	9,219	10,217	11,027	10,513	9,930	6,370
real (ms)	204,947	205,105	205,141	205,275	205,169	205,663
user (ms)	3,208	3,399	3,522	3,592	3,482	4,431
sys (ms)	411	376	335	379	379	363
LSWR-OTel-SP-LE ① (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	105,963	62,267	115,939	123,668	63,322	51,916
mean (ns)	285,806	293,675	302,573	299,168	298,552	216,285
max (ns)	950,945	1,001,696	848,895	868,464	617,033	547,116
median (ns)	291,683	296,365	303,174	303,066	303,386	219,429
std (ns)	21,489	22,560	23,863	23,759	24,203	22,381
real (ms)	208,354	208,510	208,698	208,622	208,600	211,124
user (ms)	6,079	6,195	6,368	6,265	6,220	8,808
sys (ms)	300	333	343	378	408	346
LSWR-OTel-BP-LE ① (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	23,366	26,051	30,669	19,305	13,541	15,131
mean (ns)	116,900	123,654	126,234	131,585	127,154	45,069
max (ns)	783,607	513,569	389,929	419,360	303,375	232,050
median (ns)	118,180	120,650	122,703	132,819	130,752	44,809
std (ns)	10,328	9,512	9,916	10,207	9,972	6,358
real (ms)	204,978	205,125	205,168	205,261	205,170	205,630
user (ms)	3,245	3,335	3,446	3,617	3,470	4,392
sys (ms)	379	429	384	339	362	335

LSRU-OTel-SP-LE ① (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	94,947	89,270	116,376	84,861	78,392	67,414
mean (ns)	288,689	298,138	296,374	300,658	300,299	215,586
max (ns)	957,472	889,551	766,684	759,508	683,313	479,021
median (ns)	305,975	290,830	289,137	230,047	291,786	210,677
std (ns)	22,681	23,533	22,734	23,003	23,736	22,664
real (ms)	208,483	208,658	208,638	208,737	208,739	211,175
user (ms)	6,029	6,225	6,176	6,290	6,266	8,793
sys (ms)	407	404	416	393	422	327
LRSRU-OTel-SP-LE ② (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	61,689	98,754	86,808	101,925	82,206	48,022
mean (ns)	287,596	292,763	301,050	298,985	298,528	216,613
max (ns)	1,512,586	705,500	1,620,933	730,544	605,272	483,777
median (ns)	283,274	286,731	304,177	299,669	279,096	207,857
std (ns)	23,003	22,966	24,904	23,066	23,323	19,833
real (ms)	208,473	208,580	208,739	208,699	208,696	211,242
user (ms)	5,969	6,025	6,326	6,316	6,231	8,769
sys (ms)	461	502	370	333	421	427
LSRU-OTel-BP-LE ① (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	23,063	32,129	21,435	42,594	26,186	13,692
mean (ns)	117,143	123,776	126,075	131,274	128,526	45,443
max (ns)	536,921	368,783	512,584	395,187	287,299	244,123
median (ns)	113,691	125,171	122,321	127,408	124,568	44,749
std (ns)	9,394	10,169	9,917	10,374	10,148	6,930
real (ms)	205,077	205,198	205,230	205,344	205,277	205,736
user (ms)	3,259	3,376	3,480	3,617	3,487	4,400
sys (ms)	405	406	388	367	404	358
LRSRU-OTel-BP-LE ③ (20 000 spans)						
Scenarios	1	2	3	4	5	6
min (ns)	43,641	45,021	35,309	41,164	24,339	14,637
mean (ns)	116,836	124,600	126,452	131,215	127,406	45,611
max (ns)	396,297	372,158	742,740	402,347	254,396	221,994
median (ns)	131,189	125,659	84,018	132,773	124,307	44,721
std (ns)	9,482	9,602	10,738	9,709	10,061	6,704
real (ms)	205,048	205,211	205,231	205,328	205,255	205,737
user (ms)	3,268	3,412	3,397	3,624	3,419	4,471
sys (ms)	379	396	466	352	436	326

OTel-SP-ROC ③, 289 μs for LSRU-OTel-SP-LE ①, and 288 μs for LRSRU-OTel-SP-LE ②. Utilizing LTTng reduces the wall-clock time by a factor of 10.

For the same setup, the CPU times recorded are 32316 μs (8079 μs * 20000 spans / 4000 spans), 6029 μs , and 5969 μs , respectively. With LTTng, there is a reduction in CPU time by over 80%. Interestingly, exporting spans one by one is slightly faster in a remote session than a local one. This results in a nearly 1% CPU time gain.

Comparison: NLS-OTel-BP-ROC ③ vs LSRU-OTel-BP-LE ① vs LRSRU-OTel-BP-LE ② We conduct the same comparison but with a variation. Instead of exporting spans individually, we utilize the Batching processor to group them into batches of 512 spans and we export these batches every 5s.

In all configurations, the average wall-clock time taken to export a span is $117\mu s$. However, the total CPU time recorded differs: it is 3663ms for NLS-OTel-BP-ROC ③, 3259ms for LSRU-OTel-BP-LE ① and 3268ms for LRSRU-OTEL-BP-LE ②. With LTTng, we observe a minimum 10% gain in CPU time.

Flooding OTel ring buffers The OTel batching processor uses ring buffers as a data structure to store data temporarily, before it is sent for further processing. The ring buffer is highly efficient because, when it is full, it starts overwriting the oldest data, ensuring a fixed size is maintained and resources are efficiently used.

In this experiment, we continuously export batches of spans for one minute, without any pauses, showcasing an example of ring buffers fast writer and slow reader. We then record the actual number of spans successfully exported under the three configuration groups we have been examining. The results are displayed in Table 4.4.

A significant 97% span loss rate was observed when exporting to the OTel collector, compared to a lower 64% loss rate with LTTng. This difference primarily stems from the slower speed of the OTLP Exporter used for exporting span batches to the OTel collector. This process requires an RPC call round trip, which inevitably slows down the operation. As a result, batches linger longer in the OTel ring buffers and eventually get overwritten due to the buffer fixed size.

4.5.2 Overhead of collecting application metrics

To measure the overhead associated with the collection of application metrics, we first perform an operation without collecting metrics to establish a baseline time measurement. This

Table 4.4 Flooding OTel ring buffers

Processor	NLS-OTel-ROC ③ (Exporting to remote OTel collector)	LSRU-OTel-LE ① (Exporting to local LTTng session)	LRSRU-OTEL-LE ② (Exporting to remote LTTng session)
Simple	29,275 / 29,275	1,569,145 / 1,569,145	1,546,970 / 1,546,970
Span Processor (SP)	No loss	No loss	No loss
Batching	222,219 / 7,418,929	2,863,534 / 7,829,110	2,757,642 / 7,551,265
Span Processor (BP)	(97% loss rate)	(64% loss rate)	(64% loss rate)

operation is then repeated while exporting metrics at intervals of 500 and 1000 milliseconds. The configuration used for these tests aligns with that outlined in Table 4.1, with the exception that no processor is involved.

The results for all configurations are presented in Table 4.5. In every instance, using LTTng as a metrics exporter reduces the overhead when compared to using an OTLP exporter with the OTel Collector. The maximum overhead observed with LTTng is 0.261%.

4.5.3 Overhead of collecting application logs

The overhead associated with the collection of application logs is outlined in Table 4.6. In all configuration groups, we logged a 5-bytes string 500 times, along with severity and service resource definitions such as name and version.

The process of exporting a log to an OTel collector via OTLP takes roughly 1900 microseconds, compared to approximately 230 microseconds when exporting them to LTTng. This represents an improvement factor of at least 8 times when using LTTng over OTLP.

4.5.4 Overhead of collecting host metrics

Host metrics are derived from kernel events. The overhead associated with their collection is closely related to the overhead incurred during the tracing of these kernel events. LTTng is renowned for its low overhead, attributable to its highly efficient tracing mechanisms. The actual overhead can fluctuate based on several factors, including system configuration, workload, and the specific events being traced. For accurate measurements, it is necessary to conduct a benchmark test in your particular environment. As reported in a recent survey [46], the average overhead of one kernel space tracepoint is recorded to be 92 nanoseconds.

4.5.5 Discussion and proposed tracing architecture

We have presented benchmarks illustrating the overhead of various tracing architectures utilized for the collection of application TD. As a general guideline, exporting TD using LTTng (① and ②) results in the lowest overhead. LTTng itself can be configured to transmit TD to another host (②), a feature that is slightly more efficient compared to the local export (①). We have noted up to a 1% gain. However, the recommended solution should be to log all traces locally to LTTng (①). This approach removes the need to continuously run a remote LTTng session and eliminates potential network communication issues that could arise.

Table 4.5 Application metrics collection benchmarks

Configurations	export delay (ms)	duration (ns)	duration overhead (%)	cpu time (ns)	cpu time overhead (%)
NI	500	114,541,201,139		114,537,400,000	
	1000	114,538,587,631		114,535,000,000	
NLS-OTel-LOC ③	500	115,263,640,715	0.631	115,805,500,000	1.107
	1000	114,960,380,370	0.633	115,287,200,000	0.657
NLS-OTel-ROC ③	500	115,289,853,198	0.654	115,815,700,000	1.116
	1000	115,029,764,340	0.656	115,348,000,000	0.710
NLS-OTel-LE ①	500	114,681,381,160	0.122	114,799,700,000	0.229
	1000	114,583,819,760	0.125	114,650,800,000	0.101
LSWR-OTel-LE ①	500	114,663,114,820	0.106	114,786,300,000	0.217
	1000	114,588,457,248	0.109	114,652,500,000	0.103
LSRU-OTel-LE ①	500	114,711,609,356	0.149	114,836,000,000	0.261
	1000	114,681,243,712	0.151	114,748,900,000	0.187
LRSRU-OTel-LE ②	500	114,648,509,661	0.094	114,775,500,000	0.208
	1000	114,571,802,027	0.096	114,649,600,000	0.100

Table 4.6 Application logs collection benchmarks

Configurations	min (ns)	mean (ns)	max (ns)	median (ns)	std (ns)	real (ms)	user (ms)	sys (ms)
NLS-OTel-LOC ③	162,597	1,942,427	14,075,368	1,942,680	366,043	60,257	7,179	481
NLS-OTel-ROC ③	109,952	1,933,579	12,132,325	1,856,451	341,200	60,216	7,054	581
NLS-OTel-LE ①	2,454	224,581	727,890	227,603	22,160	51,732	1,191	99
LSWR-OTel-LE ①	99,687	225,346	655,659	247,345	25,447	51,757	1,209	87
LSRU-OTel-LE ①	9,975	232,732	713,687	292,836	23,224	51,791	1,255	84
LRSRU-OTel-LE ②	90,243	230,088	716,526	242,816	27,159	51,780	1,234	89

Figure 4.4 encapsulates the proposed tracing architecture along with the boundaries of the system being traced. We distinguish between two phases: the online and offline phase. The online phase is applicable during production when the application is running. Conversely, the offline phase, which involves the analysis of collected data, is preferably conducted on a separate host at a convenient time.

4.6 USE CASE: TRACING ZEROMQ BASED MICROSERVICES

ZeroMQ [87], with its high performance and low-latency messaging capabilities, emerges as a strong choice for microservice architectures demanding real-time operation. It supports a variety of communication patterns such as request/reply, publish/subscribe, and push/pull, thus facilitating the creation of distributed applications. Its language-agnostic nature and ability to operate over multiple transport types, including TCP, in-process, inter-process, and multicast, further enhance its adaptability. Unlike traditional messaging systems, ZeroMQ does not necessitate a central message broker, thereby eliminating a potential bottleneck. It also simplifies messaging complexities, like reconnection and retries, enabling developers to focus on application logic. With scalability at its core, ZeroMQ is equipped to handle the growth inherent in microservice-based applications.

In this section, we showcase the practical applicability of our solution through a use case. We have developed a ZeroMQ client and a proxy server application in C, which we subsequently traced using our proposed solution. Given the absence of an official OTel C library, we crafted C bindings around the official C++ library. This enabled us to trace our application effectively using the newly developed C bindings library.

Figure 4.5 illustrates the application architecture. We only showcase 2 physical clients and servers, but we can use more than that. Each entity (client, server or proxy) is run isolated in a docker container.

The proxy is used to facilitate communication between clients and servers. It establishes connections on four sockets and employs a pair of ROUTER and DEALER sockets to facilitate the exchange of messages, including requests and responses, between the frontend (clients) and backend services(servers). It also uses a ROUTER socket to manage client connections, tracking the number of active connections through “CONNECT” and “DISCONNECT” messages sent by clients upon connection and disconnection respectively. Finally, a PUB socket dispatches a “KILL” message to signal a shutdown of the proxy, enabling all subscribers to terminate their activities gracefully.

Each client in the system establishes connections to two sockets. The first is a REQ socket,

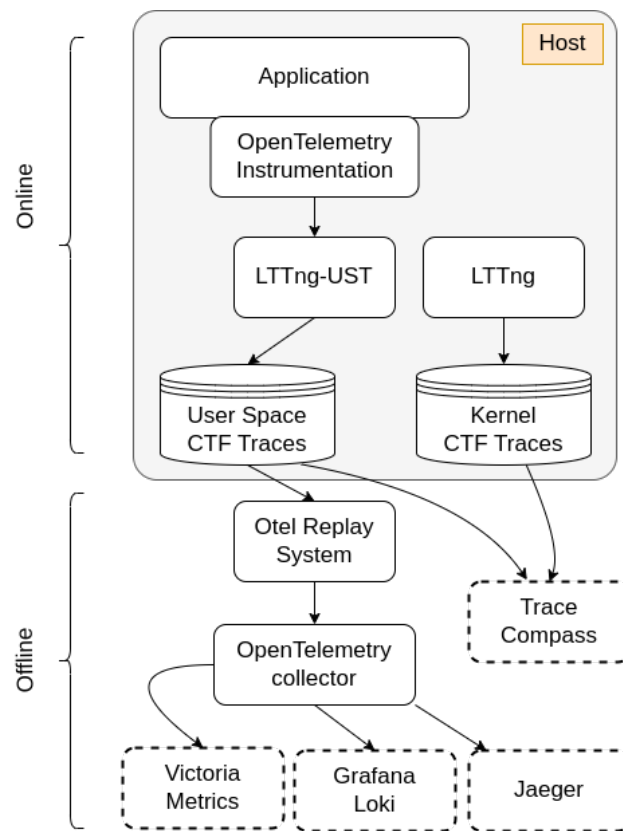


Figure 4.4 Tracing architecture

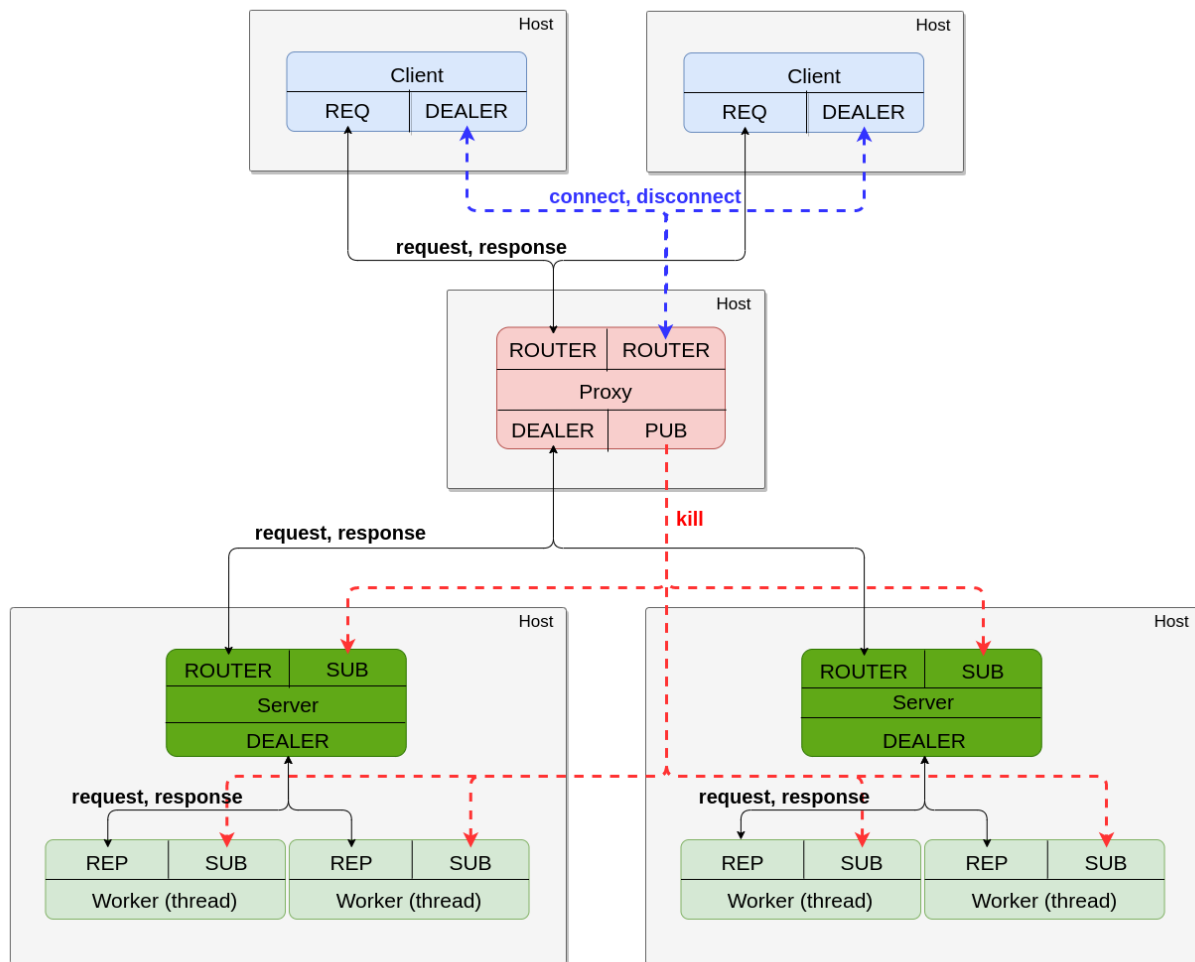


Figure 4.5 ZeroMQ client proxy server demo application

which connects to one of the proxy ROUTER sockets, facilitating the sending of requests and receiving of responses. The second is a DEALER socket, which connects to another ROUTER socket within the proxy. This socket is responsible for transmitting “CONNECT” and “DISCONNECT” messages, indicating the client connection status.

The server in our system is a multithreaded entity, with each of its threads serving as a worker responsible for handling requests. Though our illustration depicts two worker threads each, the server could support more. The server is designed to connect to three sockets for communication and operation management. A ROUTER socket interfaces with the proxy DEALER socket, enabling the transmission of requests to the workers and reception of responses from them. A SUB socket connects to the proxy PUB socket, which notifies the server when the proxy ceases operation. Lastly, a DEALER socket is bound to the workers, ensuring the forwarding of requests and replies to them. Each worker, tasked with managing the client requests, connects to two sockets. A REP socket facilitates connection to the server DEALER socket, allowing workers to receive requests and dispatch responses. Like the server, the worker also connects to a SUB socket linked to the proxy PUB socket, ensuring it is alerted when the proxy stops.

We instrumented the application to collect request traces for all client-to-server requests, application metrics for the number of messages passing through the proxy, and proxy application logs containing clients “CONNECT” and “DISCONNECT” messages. We subsequently created the replayer project (Figure 4.6), which reads the generated CTF files and sends TD to the OTel collector. The OTel collector, configured to dispatch the corresponding TD to the appropriate OBs: Jaeger for request traces, Loki for logs, and VictoriaMetrics for metrics. These OBs serve as data sources in Grafana. As a result, all TD collected can be conveniently viewed using Grafana alone.

4.7 CONCLUSIONS AND FUTURE WORK

Tracing low latency applications is a delicate task that necessitates the use of sophisticated tools. These tools should minimize the overhead introduced, to avoid disrupting the application behavior. A multitude of resources are available to aid in the gathering, visualization, and analysis of TD. Their goal is to identify and rectify any potential issues or bottlenecks that could degrade system performance, before they morph into critical problems.

In this paper, we have developed a new tracing technique, that combines state-of-the-art tools, like LTTng and OTel, to effectively collect TD from these low latency microservices. LTTng is used to collect kernel events and thereby host metrics, while OTel API and SDKs

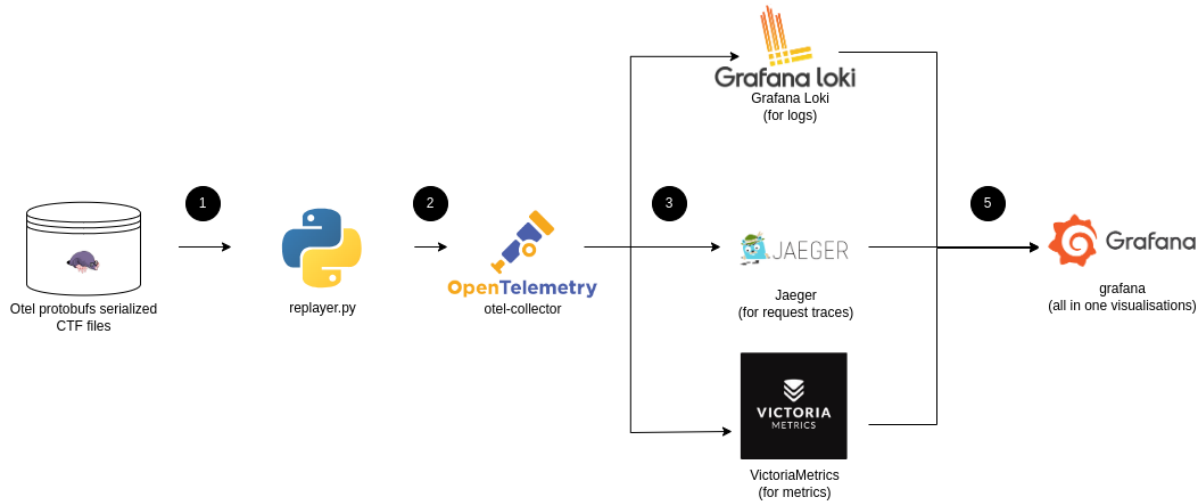


Figure 4.6 OTEL Replayer architecture

are used to instrument our microservices. The data collected, including request traces, application metrics, and logs, are later exported to LTng in high-performance binary CTF files. The results of our benchmarks showed that using LTng as an OTEL exporter can help reduce exporting latencies up to an order of magnitude, compared to all other conventional exporters such as OTLP. Additionally, we proposed a replayer architecture, which can read the generated CTF files and send TD to OBs such as Jaeger, Loki and VictoriaMetrics for in-depth analysis.

Future research should concentrate on creating analysis views specifically designed for low latency microservices, utilizing adaptable analysis tools such as Trace Compass. The addition of new Trace Compass analyses and views would eliminate the necessity for a replayer, thereby simplifying the analysis process for system administrators. This will be possible through the direct support of serialized OTEL protobufs in CTF files by Trace Compass, eliminating thereby the need for any other third-party OB. Furthermore, the development of custom-made analysis tools could be an innovative addition to the field. These tools could provide a combined TD analysis, integrating various metrics and data points, to deliver a more comprehensive understanding of the system performance. This holistic approach can potentially offer greater insights and pave the way for improved system optimization strategies.

4.8 ACKNOWLEDGMENTS

We would like to gratefully acknowledge Ericsson, Ciena, AMD, EfficiOS, NSERC and Prompt for funding this project.

CHAPTER 5 GENERAL DISCUSSION

5.1 Review of the thesis objectives

We have thoroughly addressed each element of the problem that was identified, based on the provided requirements in Section 1.2. We have proposed efficient methods for TD collection that harmonize with the resource constraints of our microservice hosts. Our article presents strategies to minimize the impact of data collection on system performance, while ensuring that the necessary data for monitoring and optimization is gathered. We have also explored ways to minimize network traffic during TD collection, thereby reducing potential system performance effects. Additionally, we have developed effective protocols for structured data collection, making subsequent analysis easier. We have also proposed techniques to analyze TD across multiple hosts to identify performance bottlenecks. Thus, our article offers comprehensive solutions to the initial problem set.

The tracing architecture that we introduced in our article successfully addresses the research objectives outlined in Section 1.3. The integration of LTTng and OTel has demonstrated its efficacy in efficiently gathering TD from low-latency microservices with limited resources. The collected TD can subsequently be examined using robust analysis tools such as Trace Compass, or observability backends including Jaeger, VictoriaMetrics, and Loki.

5.2 Comparison with other approaches

Our benchmarks unveiled the overhead associated with collecting each type of TD, using our proposed tracing architecture, as well as using other cutting-edge TD collection architectures. Our tracing system demonstrated its capability to significantly minimize the tracing overhead. In some scenarios, we observed an improvement of up to tenfold. This improvement is primarily facilitated by LTTng. Exporting TD to local files or even to a remote host using LTTng is much faster than sending TD using OTLP.

The primary distinction between our approach and others like OTel lies in the fact that our methodology is not fundamentally focused on real-time monitoring. Real-time monitoring is a continuous process of observing and analyzing a system operations, aiming to provide current information about the system performance, user activities, and overall health. However, it is important to note that achieving real-time monitoring is still possible with the use of LTTng. A remote LTTng session can be established on the target host, which will transmit all TD to a relay daemon operating on a remote server. A live trace reader can then connect to this relay

daemon and access the TD generated in real-time via the LTTng live communication protocol. Subsequently, the live trace reader can transmit the generated data to an OTel collector and execute simple procedures to read and act upon these data - including generating alerts if necessary and performing other relevant processing tasks. Since none of these actions require to modify the tracing architecture in place, the overhead on the system being traced will be similar.

CHAPTER 6 CONCLUSION

We conclude this thesis by providing a summary of the research and work conducted. Subsequently, we identify the limitations of the proposed solution and suggest potential avenues for further exploration and enhancements.

6.1 Summary of Works

Tracing low latency applications is a delicate task that necessitates the use of sophisticated tools. These tools should minimize the overhead introduced, to avoid disrupting the application behavior. A multitude of resources are available to aid in the gathering, visualization, and analysis of TD. Their goal is to identify and rectify any potential issues or bottlenecks that could degrade system performance, before they morph into critical problems. In our paper, we have developed a new tracing technique, that combines state-of-the-art tools, like LTTng and OTel, to effectively collect TD from these low latency microservices. LTTng is used to collect kernel events and thereby host metrics, while OTel API and SDKs are used to instrument our microservices. The data collected, including request traces, application metrics, and logs, are later exported to LTTng in high-performance binary CTF files. The results of our benchmarks showed that using LTTng as an OTel exporter can help reduce exporting latencies up to an order of magnitude, compared to all other conventional exporters such as OTLP. Additionally, we developed a replayer architecture, which can read the generated CTF files and send TD to OBs such as Jaeger, Loki and VictoriaMetrics for in-depth analysis.

6.2 Limitations

The implementation of our tracing architecture, which incorporates the use of a local LTTng tracing session, is capable of generating a substantial volume of data, reaching gigabytes in just a matter of minutes, particularly when numerous kernel events are activated. This scenario presents a unique challenge, considering the extensive data management requirements it brings about. Specifically, this voluminous data generation could potentially fill up storage resources, especially when dealing with a high number of microservices. One might consider transmitting all the data produced to a remote host as a possible solution. However, this approach essentially just relocates the problem, transferring the burden of handling this massive data from one system component to another. It does not fundamentally address

the issue of the substantial data generation, and it could introduce additional complications related to network bandwidth usage and data processing capabilities on the remote host.

The second limitation pertains to our reliance on the Linux Trace Toolkit Next Generation (LTTng). As a Linux-specific tool, LTTng inherently confines our solution to Linux-based systems, thereby limiting its applicability across diverse operating system environments. This constraint not only affects the scalability of our solution but also its versatility. In the modern digital ecosystem, systems often span a multitude of platforms, from traditional Windows or macOS environments to various Unix-like systems, and even to more specialized real-time or embedded operating systems. A tracing architecture that is exclusively tied to Linux may, therefore, fall short in providing a universally applicable solution. Moreover, this limitation could also impact the adoption of our solution within organizations that employ a heterogeneous mix of operating systems. Without the ability to deploy the tracing architecture across all systems uniformly, organizations might find it challenging to maintain consistency in their tracing and monitoring processes.

6.3 Future Research

While our tracing architecture presents promising results in terms of TD collection, it also underscores the need for efficient and effective storage management strategies. Future work could explore solutions such as data compression techniques, selective data collection, real-time data processing and analysis to tackle this challenge.

LTTng provides powerful tracing capabilities for Linux systems, but its platform-specific nature constitutes a significant limitation for our solution. Future work could explore potential ways to extend our architecture compatibility, such as integrating other tracing tools suitable for different operating systems, or developing a cross-platform tracing solution.

Future research should also concentrate on creating analysis views specifically designed for low latency microservices, utilizing adaptable analysis tools such as Trace Compass. The addition of new Trace Compass analyses and views would eliminate the necessity for a replayer, thereby simplifying the analysis process for system administrators. This will be possible through the direct support of serialized OTel protobufs in CTF files by Trace Compass, eliminating thereby the need for any other third-party OB. Furthermore, the development of custom-made analysis tools could be an innovative addition to the field. These tools could provide combined TD analyses, integrating various metrics and data points, to deliver a more comprehensive understanding of the system performance. This approach can potentially offer greater insights and pave the way for improved system optimization strategies.

REFERENCES

- [1] Wikipedia contributors, “Tracing (software) — Wikipedia, the free encyclopedia,” [https://en.wikipedia.org/w/index.php?title=Tracing_\(software\)&oldid=1161261031](https://en.wikipedia.org/w/index.php?title=Tracing_(software)&oldid=1161261031), 2023, [Online; accessed 18-October-2023].
- [2] “Traces.” [Online]. Available: <https://opentelemetry.io/docs/concepts/signals/traces/>
- [3] A. Anand, “What is context propagation in distributed tracing?” Apr 2023. [Online]. Available: <https://signoz.io/blog/context-propagation-in-distributed-tracing/>
- [4] Sumo Logic, “Host metrics sumo logic app - sumo logic docs,” May 2023. [Online]. Available: <https://help.sumologic.com/docs/integrations/hosts-operating-systems/host-metrics/>
- [5] The OpenTelemetry Authors, “Metrics.” [Online]. Available: <https://opentelemetry.io/docs/concepts/signals/metrics/>
- [6] J. Kanjilal, “10 application performance metrics and how to measure them - techtarget,” Jul 2022. [Online]. Available: <https://www.techtarget.com/searchapparchitecture/tip/5-application-performance-metrics-all-dev-teams-should-track>
- [7] The OpenTelemetry Authors, “Logs.” [Online]. Available: <https://opentelemetry.io/docs/concepts/signals/logs/>
- [8] —, “Observability primer.” [Online]. Available: <https://opentelemetry.io/docs/concepts/observability-primer/>
- [9] A. Janes, X. Li, and V. Lenarduzzi, “Open tracing tools: Overview and critical comparison,” no. arXiv:2207.06875, Jun 2023, arXiv:2207.06875 [cs]. [Online]. Available: <http://arxiv.org/abs/2207.06875>
- [10] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O’Reilly Media, Inc., Apr 2020, google-Books-ID: g5bcDwAAQBAJ.
- [11] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Inc., Tech. Rep., 2010. [Online]. Available: <https://research.google.com/archive/papers/dapper-2010-1.pdf>

- [12] 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*, Jun 2002, p. 595–604. [Online]. Available: <https://ieeexplore.ieee.org/document/1029005>
- [13] 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 & Implementation - Volume 6*, ser. OSDI’04. USA: USENIX Association, 2004, p. 18.
- [14] 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 & implementation*, ser. NSDI’07. USA: USENIX Association, 2007, p. 20.
- [15] Zipkin contributors, “Openzipkin · a distributed tracing system.” [Online]. Available: <https://zipkin.io/>
- [16] C. Aniszczyk, “Distributed systems tracing with zipkin,” Jun 2012. [Online]. Available: https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin
- [17] OpenTracing contributors, “Opentracing specification.” [Online]. Available: <https://opentracing.io/specification/>
- [18] N. Woods, “Opentracing joins the cloud native computing foundation,” Oct 2016. [Online]. Available: <https://www.cncf.io/blog/2016/10/11/opentracing-joins-the-cloud-native-computing-foundation/>
- [19] N. Šabić, “Opentracing: Distributed tracing’s emerging industry standard [ebook],” Mar 2019. [Online]. Available: <https://sematext.com/blog/opentracing-distributed-tracing-emerging-industry-standard/>
- [20] CNCF Staff, “Cncf archives the opentracing project,” Jan 2022. [Online]. Available: <https://www.cncf.io/blog/2022/01/31/cncf-archives-the-opentracing-project/>
- [21] The Jaeger Authors, “Jaeger: open source, distributed tracing platform.” [Online]. Available: <https://www.jaegertracing.io/>
- [22] Y. Shkuro, “Evolving distributed tracing at uber engineering,” Feb 2017. [Online]. Available: <https://www.uber.com/blog/distributed-tracing/>

- [23] The Jaeger Authors, “Jaeger architecture.” [Online]. Available: <https://www.jaegertracing.io/docs/1.32/architecture/>
- [24] Y. Shkuro, “Introducing native support for opentelemetry in jaeger,” May 2022. [Online]. Available: <https://medium.com/jaegertracing/introducing-native-support-for-opentelemetry-in-jaeger-eb661be8183c>
- [25] OpenCensus Contributors, “Opencensus.” [Online]. Available: <https://opencensus.io/>
- [26] P. Shah, “Opencensus: A stats collection and distributed tracing framework,” Jan 2018. [Online]. Available: <https://opensource.googleblog.com/2018/01/opencensus.html>
- [27] T. Young, “A roadmap to convergence,” May 2019. [Online]. Available: <https://medium.com/opentracing/a-roadmap-to-convergence-b074e5815289>
- [28] The OpenTelemetry Authors, “What is opentelemetry?” [Online]. Available: <https://opentelemetry.io/docs/what-is-opentelemetry/>
- [29] —, “Opentelemetry specification overview.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/overview/>
- [30] —, “Otlp specification 1.0.0.” [Online]. Available: <https://opentelemetry.io/docs/specs/otlp/>
- [31] —, “Opentelemetry semantic conventions 1.21.0.” [Online]. Available: <https://opentelemetry.io/docs/specs/semconv/>
- [32] —, “Propagators api.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/context/api-propagators/>
- [33] —, “Baggage api.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/baggage/api/>
- [34] —, “Tracing api.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/trace/api/>
- [35] —, “Metrics api.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/metrics/api/>
- [36] —, “Logs bridge api.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/logs/bridge-api/>

- [37] —, “Events api interface.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/logs/event-api/>
- [38] —, “Resource sdk.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/resource/sdk/>
- [39] —, “Logs sdk.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/logs/sdk/>
- [40] —, “Metrics sdk.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/metrics/sdk/>
- [41] —, “Tracing sdk.” [Online]. Available: <https://opentelemetry.io/docs/specs/otel/trace/sdk/>
- [42] —, “Opentelemetry automatic instrumentation.” [Online]. Available: <https://opentelemetry.io/docs/concepts/instrumentation/automatic/>
- [43] —, “Opentelemetry collector.” [Online]. Available: <https://opentelemetry.io/docs/collector/>
- [44] T. Bird, “Measuring function duration with ftrace,” in *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2009.
- [45] A. Aranya, “Tracefs: A file system to trace them all,” in *3rd USENIX Conference on File and Storage Technologies (FAST 04)*. San Francisco, CA: USENIX Association, mar 2004. [Online]. Available: <https://www.usenix.org/conference/fast-04/tracefs-file-system-trace-them-all>
- [46] M. Gebai and M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead,” *ACM Computing Surveys*, vol. 51, no. 2, pp. 26:1–26:33, Mar 2018. [Online]. Available: <https://dl.acm.org/doi/10.1145/3158644>
- [47] LTTng Authors, “Lttng: an open source tracing framework for linux.” [Online]. Available: <https://lttng.org/>
- [48] M. Desnoyers and M. R. Dagenais, “The lttng tracer: A low impact performance and behavior monitor for gnu/linux,” in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.
- [49] —, “Lockless multi-core high-throughput buffering scheme for kernel tracing,” *ACM SIGOPS Operating Systems Review*, vol. 46, no. 3, p. 65–81, 2012. [Online]. Available: <https://dl.acm.org/doi/10.1145/2421648.2421659>

- [50] LTTng Authors, “Lttng v2.13 — lttng documentation.” [Online]. Available: <https://lttng.org/docs/v2.13/>
- [51] CTF Authors, “Common trace format v1.8.3.” [Online]. Available: <https://diamon.org/ctf/>
- [52] J. Blunck, M. Desnoyers, and P.-M. Fournier, “Userspace application tracing with markers and tracepoints,” in *Proceedings of the Linux Kongress*, 2009, pp. 7–14.
- [53] “perf: Linux profiling with performance counters.” [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page
- [54] The kernel development community, “Perf events and tool security — the linux kernel documentation.” [Online]. Available: <https://www.kernel.org/doc/html/v5.7/admin-guide/perf-security.html#perf-events-perf-unprivileged-users>
- [55] A. R. Ghods, “A study of linux perf and slab allocation sub-systems,” Master’s thesis, University of Waterloo, 2016.
- [56] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, “Creating complex network services with ebpf: Experience and lessons learned,” in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2018, pp. 1–8. [Online]. Available: <https://ieeexplore.ieee.org/document/8850758>
- [57] F. C. Eigler and R. Hat, “Problem solving with systemtap,” in *Proc. of the Ottawa Linux Symposium*. Citeseer, 2006, pp. 261–268.
- [58] C. Vitaly, “strace - trace system calls and signals.” [Online]. Available: <https://strace.io/>
- [59] Prometheus Authors, “Monitoring linux host metrics with the node exporter - prometheus.” [Online]. Available: <https://prometheus.io/docs/guides/node-exporter/>
- [60] The FluentBit Authors, “Fluentd & fluent bit.” [Online]. Available: <https://docs.fluentbit.io/manual/about/fluentd-and-fluent-bit>
- [61] [Online]. Available: <https://www.elastic.co/fr/beats>
- [62] New Relic, “Compatibility and requirements for new relic agents and products - new relic documentation.” [Online]. Available: <https://docs.newrelic.com/docs/new-relic-solutions/new-relic-one/install-configure/compatibility-requirements-new-relic-agents-products/>

- [63] AWS, “Collect metrics, logs, and traces from amazon ec2 instances and on-premises servers with the cloudwatch agent - amazon cloudwatch.” [Online]. Available: <https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Install-CloudWatch-Agent.html>
- [64] Google Cloud, “Ops agent overview - cloud monitoring - google cloud.” [Online]. Available: <https://cloud.google.com/monitoring/agent/ops-agent>
- [65] Azure Monitor, “Azure monitor agent overview - azure monitor,” Oct 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-monitor/agents/agents-overview>
- [66] T. Davidson, E. Wall, and J. Mace, “A qualitative interview study of distributed tracing visualisation: A characterisation of challenges and opportunities,” *IEEE Transactions on Visualization and Computer Graphics*, p. 1–12, 2023. [Online]. Available: <https://ieeexplore.ieee.org/document/10034850>
- [67] The Apache Software Foundation, “Apache skywalking.” [Online]. Available: <https://skywalking.apache.org/>
- [68] A. Bader, “Comparison of time series databases,” 2016. [Online]. Available: <http://elib.uni-stuttgart.de/handle/11682/9793>
- [69] Prometheus, “Prometheus - monitoring system & time series database.” [Online]. Available: <https://prometheus.io/>
- [70] VictoriaMetrics, “Victoriametrics: Simple & reliable monitoring for everyone.” [Online]. Available: <https://victoriametrics.com/>
- [71] Prometheus, “Prometheus pushgateway.” [Online]. Available: <https://github.com/prometheus/pushgateway/blob/361849a107a59026c8a3b9a2ebc6e9e2638cbe4f/README.md#about-timestamps>
- [72] Eclipse, “Trace compass.” [Online]. Available: <https://eclipse.dev/tracecompass/>
- [73] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers, “Recovering system metrics from kernel trace,” Ottawa, Canada, 2011. [Online]. Available: <https://www.kernel.org/doc/ols/2011/ols2011-giraldeau.pdf>
- [74] L. Gelle, N. Ezzati-Jivan, and M. R. Dagenais, “Combining distributed and kernel tracing for performance analysis of cloud applications,” *Electronics*, vol. 10, no. 21, 2021. [Online]. Available: <https://www.mdpi.com/2079-9292/10/21/2610>

- [75] G. Labs, “Grafana: The open observability platform.” [Online]. Available: <https://grafana.com/>
- [76] I. Kohyarnejadfard, “System performance anomaly detection using tracing data analysis,” phd, Polytechnique Montréal, Apr 2022. [Online]. Available: <https://publications.polymtl.ca/10281/>
- [77] Q. Fournier, “Machine learning for anomaly detection in kernel traces,” phd, Polytechnique Montréal, Dec 2022. [Online]. Available: <https://publications.polymtl.ca/10708/>
- [78] S. Agrawal and J. Agrawal, “Survey on anomaly detection using data mining techniques,” *Procedia Computer Science*, vol. 60, p. 708–713, Jan. 2015. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050915023479>
- [79] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, 2009. [Online]. Available: <https://doi.org/10.1145/1541880.1541882>
- [80] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, “Performance anomaly detection and bottleneck identification,” *ACM Computing Surveys*, vol. 48, no. 1, pp. 4:1–4:35, 2015. [Online]. Available: <https://doi.org/10.1145/2791120>
- [81] A. B. Nassif, M. A. Talib, Q. Nasir, and F. M. Dakalbab, “Machine learning for anomaly detection: A systematic review,” *IEEE Access*, vol. 9, p. 78658–78700, 2021. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9439459>
- [82] T. Zach, “Jaeger tracing: The ultimate guide,” Feb 2022. [Online]. Available: <https://www.aspecto.io/blog/jaeger-tracing-the-ultimate-guide/>
- [83] L. Gelle, “Traçage noyau et distribué d’applications réparties pour l’analyse de la performance,” masters, Polytechnique Montréal, Aug 2019. [Online]. Available: <https://publications.polymtl.ca/3996/>
- [84] EfficiOS, “Babeltrace: A trace manipulation toolkit.” [Online]. Available: <https://babeltrace.org/>
- [85] Twitter, “Announcing snowflake.” [Online]. Available: https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake
- [86] The OpenTelemetry Authors, “Performance benchmark of opentelemetry api,” <https://opentelemetry.io/docs/specs/otel/performance-benchmark/>.

- [87] P. Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

APPENDIX A PROTOBUF AND GRPC DEFINITIONS FOR OPENTELEMETRY REQUEST TRACES

Request trace service proto definition

The listing A.1 is a trace service definition of the official `opentelemetry-proto` repository. This Service can be used to push spans between one Application instrumented with OpenTelemetry and a collector, or between a collector and a central collector (in this case spans are sent/received to/from multiple Applications).

The service defines one gRPC remote procedure call: `Export`. This remote procedure is invoked with `ExportTraceServiceRequest` which is essentially holds spans.

Listing A.1 OpenTelemetry proto definition of the request trace service

```

1 // Copyright 2019, OpenTelemetry Authors
2 //
3 // Licensed under the Apache License, Version 2.0 (the "License");
4 // you may not use this file except in compliance with the License.
5 // You may obtain a copy of the License at
6 //
7 //     http://www.apache.org/licenses/LICENSE-2.0
8 //
9 // Unless required by applicable law or agreed to in writing, software
10 // distributed under the License is distributed on an "AS IS" BASIS,
11 // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or ←
12 //   implied.
13 // See the License for the specific language governing permissions and
14 // limitations under the License.
15
16
17 syntax = "proto3";
18
19 package opentelemetry.proto.collector.trace.v1;
20
21 import "opentelemetry/proto/trace/v1/trace.proto";
22
23 option csharp_namespace = "OpenTelemetry.Proto.Collector.Trace.V1";
24 option java_multiple_files = true;
25 option java_package = "io.opentelemetry.proto.collector.trace.v1";
26 option java_outer_classname = "TraceServiceProto";
27 option go_package = "go.opentelemetry.io/proto/otlp/collector/trace/v1";

```

```

26
27 // Service that can be used to push spans between one Application ←
    instrumented with
28 // OpenTelemetry and a collector, or between a collector and a central ←
    collector (in this
29 // case spans are sent/received to/from multiple Applications).
30 service TraceService {
31     // For performance reasons, it is recommended to keep this RPC
32     // alive for the entire life of the application.
33     rpc Export(ExportTraceServiceRequest) returns (←
        ExportTraceServiceResponse) {}
34 }
35
36 message ExportTraceServiceRequest {
37     // An array of ResourceSpans.
38     // For data coming from a single resource this array will typically ←
        contain one
39     // element. Intermediary nodes (such as OpenTelemetry Collector) that ←
        receive
40     // data from multiple origins typically batch the data before ←
        forwarding further and
41     // in that case this array will contain multiple elements.
42     repeated opentelemetry.proto.trace.v1.ResourceSpans resource_spans = 1;
43 }
44
45 message ExportTraceServiceResponse {
46     // The details of a partially successful export request.
47     //
48     // If the request is only partially accepted
49     // (i.e. when the server accepts only parts of the data and rejects the ←
        rest)
50     // the server MUST initialize the `partial_success` field and MUST
51     // set the `rejected_<signal>` with the number of items it rejected.
52     //
53     // Servers MAY also make use of the `partial_success` field to convey
54     // warnings/suggestions to senders even when the request was fully ←
        accepted.
55     // In such cases, the `rejected_<signal>` MUST have a value of `0` and
56     // the `error_message` MUST be non-empty.
57     //
58     // A `partial_success` message with an empty value (rejected_<signal> = ←
        0 and
59     // `error_message` = "") is equivalent to it not being set/present. ←
        Senders

```

```

60 // SHOULD interpret it the same way as in the full success case.
61 ExportTracePartialSuccess partial_success = 1;
62 }
63
64 message ExportTracePartialSuccess {
65 // The number of rejected spans.
66 //
67 // A `rejected_<signal>` field holding a `0` value indicates that the
68 // request was fully accepted.
69 int64 rejected_spans = 1;
70
71 // A developer-facing human-readable message in English. It should be ←
    used
72 // either to explain why the server rejected parts of the data during a ←
    partial
73 // success or to convey warnings/suggestions during a full success. The ←
    message
74 // should offer guidance on how users can address such issues.
75 //
76 // error_message is an optional field. An error_message with an empty ←
    value
77 // is equivalent to it not being set.
78 string error_message = 2;
79 }

```

Request trace proto definition

The listing A.2 contains all protobuf definitions related to request traces data from the official opentelemetry-proto repository.

Listing A.2 OpenTelemetry proto definition of a request trace

```

1 // Copyright 2019, OpenTelemetry Authors
2 //
3 // Licensed under the Apache License, Version 2.0 (the "License");
4 // you may not use this file except in compliance with the License.
5 // You may obtain a copy of the License at
6 //
7 //     http://www.apache.org/licenses/LICENSE-2.0
8 //
9 // Unless required by applicable law or agreed to in writing, software
10 // distributed under the License is distributed on an "AS IS" BASIS,

```

```

11 // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or ↵
    implied.
12 // See the License for the specific language governing permissions and
13 // limitations under the License.
14
15 syntax = "proto3";
16
17 package opentelemetry.proto.trace.v1;
18
19 import "opentelemetry/proto/common/v1/common.proto";
20 import "opentelemetry/proto/resource/v1/resource.proto";
21
22 option csharp_namespace = "OpenTelemetry.Proto.Trace.V1";
23 option java_multiple_files = true;
24 option java_package = "io.opentelemetry.proto.trace.v1";
25 option java_outer_classname = "TraceProto";
26 option go_package = "go.opentelemetry.io/proto/otlp/trace/v1";
27
28 // TracesData represents the traces data that can be stored in a ↵
    persistent storage,
29 // OR can be embedded by other protocols that transfer OTLP traces data ↵
    but do
30 // not implement the OTLP protocol.
31 //
32 // The main difference between this message and collector protocol is ↵
    that
33 // in this message there will not be any "control" or "metadata" specific↵
    to
34 // OTLP protocol.
35 //
36 // When new fields are added into this message, the OTLP request MUST be ↵
    updated
37 // as well.
38 message TracesData {
39     // An array of ResourceSpans.
40     // For data coming from a single resource this array will typically ↵
        contain
41     // one element. Intermediary nodes that receive data from multiple ↵
        origins
42     // typically batch the data before forwarding further and in that case ↵
        this
43     // array will contain multiple elements.
44     repeated ResourceSpans resource_spans = 1;
45 }

```

```

46
47 // A collection of ScopeSpans from a Resource.
48 message ResourceSpans {
49     reserved 1000;
50
51     // The resource for the spans in this message.
52     // If this field is not set then no resource info is known.
53     opentelemetry.proto.resource.v1.Resource resource = 1;
54
55     // A list of ScopeSpans that originate from a resource.
56     repeated ScopeSpans scope_spans = 2;
57
58     // This schema_url applies to the data in the "resource" field. It does↵
59     // not apply
60     // to the data in the "scope_spans" field which have their own ↵
61     // schema_url field.
62     string schema_url = 3;
63 }
64
65 // A collection of Spans produced by an InstrumentationScope.
66 message ScopeSpans {
67     // The instrumentation scope information for the spans in this message.
68     // Semantically when InstrumentationScope isn't set, it is equivalent ↵
69     // with
70     // an empty instrumentation scope name (unknown).
71     opentelemetry.proto.common.v1.InstrumentationScope scope = 1;
72
73     // A list of Spans that originate from an instrumentation scope.
74     repeated Span spans = 2;
75
76     // This schema_url applies to all spans and span events in the "spans" ↵
77     // field.
78     string schema_url = 3;
79 }
80
81 // A Span represents a single operation performed by a single component ↵
82 // of the system.
83 //
84 // The next available field id is 17.
85 message Span {
86     // A unique identifier for a trace. All spans from the same trace share
87     // the same `trace_id`. The ID is a 16-byte array. An ID with all ↵
88     // zeroes
89     // is considered invalid.

```

```

84  //
85  // This field is semantically required. Receiver should generate new
86  // random trace_id if empty or invalid trace_id was received.
87  //
88  // This field is required.
89  bytes trace_id = 1;
90
91  // A unique identifier for a span within a trace, assigned when the ↵
    span
92  // is created. The ID is an 8-byte array. An ID with all zeroes is ↵
    considered
93  // invalid.
94  //
95  // This field is semantically required. Receiver should generate new
96  // random span_id if empty or invalid span_id was received.
97  //
98  // This field is required.
99  bytes span_id = 2;
100
101  // trace_state conveys information about request position in multiple ↵
    distributed tracing graphs.
102  // It is a trace_state in w3c-trace-context format: https://www.w3.org/↵
    TR/trace-context/#tracestate-header
103  // See also https://github.com/w3c/distributed-tracing for more details↵
    about this field.
104  string trace_state = 3;
105
106  // The `span_id` of this span's parent span. If this is a root span, ↵
    then this
107  // field must be empty. The ID is an 8-byte array.
108  bytes parent_span_id = 4;
109
110  // A description of the span's operation.
111  //
112  // For example, the name can be a qualified method name or a file name
113  // and a line number where the operation is called. A best practice is ↵
    to use
114  // the same display name at the same call point in an application.
115  // This makes it easier to correlate spans in different traces.
116  //
117  // This field is semantically required to be set to non-empty string.
118  // Empty value is equivalent to an unknown span name.
119  //
120  // This field is required.

```

```

121 string name = 5;
122
123 // SpanKind is the type of span. Can be used to specify additional ↵
    relationships between spans
124 // in addition to a parent/child relationship.
125 enum SpanKind {
126     // Unspecified. Do NOT use as default.
127     // Implementations MAY assume SpanKind to be INTERNAL when receiving ↵
    UNSPECIFIED.
128     SPAN_KIND_UNSPECIFIED = 0;
129
130     // Indicates that the span represents an internal operation within an ↵
    application,
131     // as opposed to an operation happening at the boundaries. Default ↵
    value.
132     SPAN_KIND_INTERNAL = 1;
133
134     // Indicates that the span covers server-side handling of an RPC or ↵
    other
135     // remote network request.
136     SPAN_KIND_SERVER = 2;
137
138     // Indicates that the span describes a request to some remote service ↵
    .
139     SPAN_KIND_CLIENT = 3;
140
141     // Indicates that the span describes a producer sending a message to ↵
    a broker.
142     // Unlike CLIENT and SERVER, there is often no direct critical path ↵
    latency relationship
143     // between producer and consumer spans. A PRODUCER span ends when the ↵
    message was accepted
144     // by the broker while the logical processing of the message might ↵
    span a much longer time.
145     SPAN_KIND_PRODUCER = 4;
146
147     // Indicates that the span describes consumer receiving a message ↵
    from a broker.
148     // Like the PRODUCER kind, there is often no direct critical path ↵
    latency relationship
149     // between producer and consumer spans.
150     SPAN_KIND_CONSUMER = 5;
151 }
152

```

```

153 // Distinguishes between spans generated in a particular context. For ↵
    example,
154 // two spans with the same name may be distinguished using `CLIENT` (↵
    caller)
155 // and `SERVER` (callee) to identify queueing latency associated with ↵
    the span.
156 SpanKind kind = 6;
157
158 // start_time_unix_nano is the start time of the span. On the client ↵
    side, this is the time
159 // kept by the local machine where the span execution starts. On the ↵
    server side, this
160 // is the time when the server's application handler starts running.
161 // Value is UNIX Epoch time in nanoseconds since 00:00:00 UTC on 1 ↵
    January 1970.
162 //
163 // This field is semantically required and it is expected that end_time↵
    >= start_time.
164 fixed64 start_time_unix_nano = 7;
165
166 // end_time_unix_nano is the end time of the span. On the client side, ↵
    this is the time
167 // kept by the local machine where the span execution ends. On the ↵
    server side, this
168 // is the time when the server application handler stops running.
169 // Value is UNIX Epoch time in nanoseconds since 00:00:00 UTC on 1 ↵
    January 1970.
170 //
171 // This field is semantically required and it is expected that end_time↵
    >= start_time.
172 fixed64 end_time_unix_nano = 8;
173
174 // attributes is a collection of key/value pairs. Note, global ↵
    attributes
175 // like server name can be set using the resource API. Examples of ↵
    attributes:
176 //
177 //     "/http/user_agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_1↵
        4_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 ↵
        Safari/537.36"
178 //     "/http/server_latency": 300
179 //     "abc.com/myattribute": true
180 //     "abc.com/score": 10.239
181 //

```

```

182 // The OpenTelemetry API specification further restricts the allowed ↵
    value types:
183 // https://github.com/open-telemetry/opentelemetry-specification/blob/↵
    main/specification/common/README.md#attribute
184 // Attribute keys MUST be unique (it is not allowed to have more than ↵
    one
185 // attribute with the same key).
186 repeated opentelemetry.proto.common.v1.KeyValue attributes = 9;
187
188 // dropped_attributes_count is the number of attributes that were ↵
    discarded. Attributes
189 // can be discarded because their keys are too long or because there ↵
    are too many
190 // attributes. If this value is 0, then no attributes were dropped.
191 uint32 dropped_attributes_count = 10;
192
193 // Event is a time-stamped annotation of the span, consisting of user-↵
    supplied
194 // text description and key-value pairs.
195 message Event {
196     // time_unix_nano is the time the event occurred.
197     fixed64 time_unix_nano = 1;
198
199     // name of the event.
200     // This field is semantically required to be set to non-empty string.
201     string name = 2;
202
203     // attributes is a collection of attribute key/value pairs on the ↵
    event.
204     // Attribute keys MUST be unique (it is not allowed to have more than ↵
    one
205     // attribute with the same key).
206     repeated opentelemetry.proto.common.v1.KeyValue attributes = 3;
207
208     // dropped_attributes_count is the number of dropped attributes. If ↵
    the value is 0,
209     // then no attributes were dropped.
210     uint32 dropped_attributes_count = 4;
211 }
212
213 // events is a collection of Event items.
214 repeated Event events = 11;
215
216 // dropped_events_count is the number of dropped events. If the value ↵

```

```

    is 0, then no
217 // events were dropped.
218 uint32 dropped_events_count = 12;
219
220 // A pointer from the current span to another span in the same trace or↵
    in a
221 // different trace. For example, this can be used in batching ↵
    operations,
222 // where a single batch handler processes multiple requests from ↵
    different
223 // traces or when the handler receives a request from a different ↵
    project.
224 message Link {
225     // A unique identifier of a trace that this linked span is part of. ↵
    The ID is a
226     // 16-byte array.
227     bytes trace_id = 1;
228
229     // A unique identifier for the linked span. The ID is an 8-byte array↵
    .
230     bytes span_id = 2;
231
232     // The trace_state associated with the link.
233     string trace_state = 3;
234
235     // attributes is a collection of attribute key/value pairs on the ↵
    link.
236     // Attribute keys MUST be unique (it is not allowed to have more than↵
    one
237     // attribute with the same key).
238     repeated opentelemetry.proto.common.v1.KeyValue attributes = 4;
239
240     // dropped_attributes_count is the number of dropped attributes. If ↵
    the value is 0,
241     // then no attributes were dropped.
242     uint32 dropped_attributes_count = 5;
243 }
244
245 // links is a collection of Links, which are references from this span ↵
    to a span
246 // in the same or different trace.
247 repeated Link links = 13;
248
249 // dropped_links_count is the number of dropped links after the maximum↵

```

```

        size was
250 // enforced. If this value is 0, then no links were dropped.
251 uint32 dropped_links_count = 14;
252
253 // An optional final status for this span. Semantically when Status isn't
    't set, it means
254 // span's status code is unset, i.e. assume STATUS_CODE_UNSET (code = 0).
255 Status status = 15;
256 }
257
258 // The Status type defines a logical error model that is suitable for
    different
259 // programming environments, including REST APIs and RPC APIs.
260 message Status {
261     reserved 1;
262
263     // A developer-facing human readable error message.
264     string message = 2;
265
266     // For the semantics of status codes see
267     // https://github.com/open-telemetry/opentelemetry-specification/blob/main/
        specification/trace/api.md#set-status
268     enum StatusCode {
269         // The default status.
270         STATUS_CODE_UNSET = 0;
271         // The Span has been validated by an Application developers or
        Operator to have
272         // completed successfully.
273         STATUS_CODE_OK = 1;
274         // The Span contains an error.
275         STATUS_CODE_ERROR = 2;
276     };
277
278     // The status code.
279     StatusCode code = 3;
280 }

```

APPENDIX B OPENTELEMETRY SEMANTIC CONVENTIONS FOR SERVERS GENERAL ATTRIBUTES

The semantic conventions general attributes described are not specific to a particular operation but rather generic. They may be used in any Span they apply to. The table B.1 is an exemple of general attributes for servers.

Table B.1 OpenTelemetry semantic conventions for servers general attributes

Attribute	Type	Description	Examples	Requirement Level
<i>server.address</i>	string	Server address - domain name if available without reverse DNS lookup, otherwise IP address or Unix domain socket name. [1]	<i>example.com</i> ; <i>10.1.2.80</i> ; <i>/tmp/my.sock</i>	Recommended
<i>server.port</i>	int	Server port number [2]	80; 8080; 443	Recommended

[1]: When observed from the client side, and when communicating through an intermediary, *server.address* SHOULD represent the server address behind any intermediaries (e.g. proxies) if it's available.

[2]: When observed from the client side, and when communicating through an intermediary, *server.port* SHOULD represent the server port behind any intermediaries (e.g. proxies) if it's available.

server.address and *server.port* represent logical server name and port. Semantic conventions that refer to these attributes SHOULD specify what these attributes mean in their context.