Nahida: In-Band Distributed Tracing with eBPF

Wanqi Yang
Sun Yat-sen University
yangwq23@mail2.sysu.edu.cn

Pengfei Chen Sun Yat-sen University chenpf7@mail.sysu.edu.cn Kai Liu
Alibaba Group
qianlu.lk@alibaba-inc.com

Huxing Zhang
Alibaba Group
huxing.zhx@alibaba-inc.com

Abstract

Microservices are commonly used in modern cloud-native applications to achieve agility. However, the complexity of service dependencies in large-scale microservices systems can lead to anomaly propagation, making fault troubleshooting a challenge. To address this issue, distributed tracing systems have been proposed to trace complete request execution paths, enabling developers to troubleshoot anomalous services. However, existing distributed tracing systems have limitations such as invasive instrumentation, trace loss, or inaccurate trace correlation. To overcome these limitations, we propose a new tracing system based on eBPF (extended Berkeley Packet Filter), named Nahida, that can track complete requests in the kernel without intrusion, regardless of programming language or implementation. Our evaluation results show that Nahida can track over 92% of requests with stable accuracy, even under the high concurrency of user requests, while the state-of-the-art non-invasive approaches can not track any of the requests. Importantly, Nahida can track requests served by a multi-threaded application that none of the existing invasive tracing systems can handle by instrumenting tracing codes into libraries. Moreover, the overhead introduced by Nahida is negligible, increasing service latency by only 1.55%–2.1%. Overall, Nahida provides an effective and non-invasive solution for distributed tracing.

1 Introduction

Modern large-scale distributed systems are often implemented in microservices architecture due to high scalability and reliability [15, 29, 36–38]. However, managing such systems can be challenging, especially when they fail. The complexity of service dependencies and anomaly propagation in such systems make troubleshooting difficult. To address this issue, distributed tracing systems [2, 10, 17, 20, 23, 30, 31, 35, 40] have been proposed to help developers and operators troubleshoot anomalous services, and are now widely used in the production environment.

Distributed tracing systems track the end-to-end requests, recording calls from downstream services to upstream services along with their duration. With tracing data, developers and operators can easily troubleshoot anomalous services, even when anomaly propagation occurs, thereby facilitating system failure resolution. However, tracing end-to-end requests in a large-scale distributed system can be difficult, as in such systems, an end-to-end request often spans multiple services across different physical machines and geographical locations, where services may be implemented in different high-level languages and communicate in different protocols.

Distributed tracing systems are mainly chosen to be implemented in an invasive manner to meet the challenges above, such as Zipkin [40], Jaeger [23], OpenTelemetry [30], Open-Tracing [31], X-Trace [20] and Dapper [35]. These invasive tracing systems require developers to weave tracing code into application source code or dependent third-party libraries, thus automatically generating traces as the application runs. However, these invasive tracing systems have their downside including requiring explicit and manual tracing code instrumentation and lacking support for hot updates. The frequent updates to applications and dependent third-party libraries require developers to spend significant time repeatedly instrumenting tracing code to each new version of the library and application to ensure accurate tracing functionality. Furthermore, the addition of tracing code to the libraries may result in version incompatibilities, and potentially introduce security risks, program bugs, or fatal errors. To eliminate code staking for distributed tracing, non-invasive distributed tracing systems are explored to track end-to-end requests without any instrumentation into application source codes. Existing noninvasive tracing systems use aggregation algorithms based on commonly-used network metrics and metadata [9, 12, 32, 34]. However, these non-invasive implementations fail to reconstruct request causality and track end-to-end requests, making them impractical in the real-world scenario with highly concurrent requests.

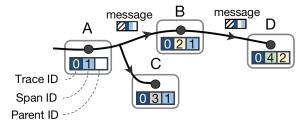
To implement a practical non-intrusive tracing system that can track end-to-end requests of various applications and handle high concurrent requests, we will face the following challenges.

- Tracking complex end-to-end requests: In a largescale distributed system, multiple services have to coordinate and cooperate to serve a user request, resulting in complex request causality. It is difficult for non-invasive tracing systems to refactor request causality without instrumentation in source codes.
- Compatible with heterogeneous high-level languages:
 Modern distributed systems are built in multiple high-level languages, such as Java, Python, and Golang. A non-invasive tracing system should be compatible with those commonly used high-level languages.
- Supporting single-threaded and multi-threaded applications: Applications serve requests in either single-threaded or multi-threaded manners. Tracing systems should adapt to different implementations of applications and track accurate end-to-end requests for them.
- Maintaining high performance: Non-invasive tracing systems must maintain high performance even in largescale distributed systems. The overhead of tracing should be minimized and not become a system bottleneck.

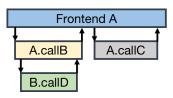
To meet the challenges mentioned above, we propose a noninvasive tracing system, named Nahida, which is based on eBPF (extended Berkeley Packet Filter) [13,16], a technology that allows user programs to be executed in the Linux kernel. Nahida first intercepts requests and responses with eBPF and injects custom trace contexts (e.g., trace ID) as tags into the messages. These contexts travel along with requests across the network from the downstream to the upstream service, and can be automatically injected into the subsequent requests without user awareness. Trace contexts are used to track endto-end requests even in complex large-scale systems. Nahida implements context injection and extraction in Linux kernel, which are decoupled from user application implementations. This allows Nahida to track end-to-end requests for applications implemented in multiple high-level languages, including Java, Python, Golang, etc. Our evaluation results demonstrate that Nahida can track more than 92% of end-to-end requests for single-threaded and multi-threaded microservices applications, even under high concurrency. Moreover, Nahida has a little impact on service performance, such as a 1.55%-2.1% increase in service latency, an additional 1%-3.78% of CPU and memory consumption.

Contributions of this paper can be summarized as follows.

We have implemented an in-kernel non-invasive distributed tracing system, Nahida, based on eBPF to successfully track end-to-end requests in large-scale distributed systems, regardless of application implementations in heterogeneous high-level languages and encapsulation in containers.



(a) An end-to-end request to the application.



(b) A generated trace tree.

Figure 1: An example trace of a distributed application.

- We implement trace context propagation in the kernel to track end-to-end requests. With in-kernel context propagation, Nahida can adapt to single-threaded and multithreaded application implementations. In contrast, existing invasive tracing systems fail in tracking end-to-end requests for multi-threaded applications with instrumentation in dependent third-party libraries, and so do the existing non-invasive tracing systems.
- We conduct extensive experiments and successfully deploy Nahida in the real-world environment. Experiment results show that Nahida can achieve high tracing accuracy of more than 92% with an insignificant overhead of 1.55%–2.1% latency increasing.

2 Background

2.1 Distributed Tracing System

Distributed tracing systems provide developers with the ability to track end-to-end requests which flow through a complex distributed system consisting of multiple microservices [10, 20, 35]. By capturing and storing the request information, distributed tracing systems can offer insights into the performance of each service and its contribution to the overall system performance. Typically, information of an end-to-end request is displayed in a trace tree, which visualizes the path of a request as it travels through the system and the time each service takes to process the request. Analyzing trace trees can help developers identify bottlenecks and troubleshoot performance issues related to the request. As distributed systems and microservices architectures continue to gain in popularity, distributed tracing systems have become a crucial tool for managing and monitoring complex systems.

The following concepts are helpful in understanding distributed tracing systems:

- Trace: In a tracing system, a trace represents the complete request path for an end-to-end request. Each trace is identified by a unique *trace ID*, which is used to correlate spans of the same trace across different services.
- Span: A span represents an operation in a trace, for example, service A calls service B. Each span is identified by a unique *span ID* and is recorded with a start and end timestamp.
- Parent Span: A span can have a parent span, which
 indicates their causal relationship. The parent span is
 identified by a parent ID, and helps to show the sequence
 of operations in an end-to-end request.
- Root Span: The root span is the first span in a trace and has no parent span. It represents the entry of the request.
- Trace Context: The trace context contains the trace ID, span ID, parent ID, and other relevant information. It is passed along with transmitting messages among services and enables associating two causal requests.

Figure 1 shows an example to illustrate how a distributed tracing system works to help with failure troubleshooting. The distributed tracing system can collect trace contexts with the same trace ID from four services of an end-to-end request to an application, and then construct a trace tree with four spans as shown in Figure 1(b). When a failure in service D happens, traveling through the trace tree from the root span with ID 1 to its children span 2 and then span 4, the failure can be troubleshot as soon as possible by comparing the serving duration of each span. This example highlights the importance of propagating trace contexts and how distributed tracing systems enable efficient troubleshooting in large-scale distributed systems.

2.2 eBPF

BPF (Berkeley Packet Filter) [28] was initially designed for packet filtering and later extended as eBPF (extended BPF) [13,16] to enable user-defined event-triggered programs to be executed in the Linux kernel. eBPF provides a set of hooks in the network, where eBPF programs can be attached, enabling better network functionality [1,3,11,18,24,33,39].

The kernel programmability provided by eBPF makes it possible to implement a non-invasive distributed tracing system. Nahida can attach eBPF programs on system calls and sockets to track end-to-end requests without instrumenting applications or modifying the Linux kernel. Moreover, the key-value datastore called eBPF Maps, defined by eBPF, helps our tracing system store data in the kernel. For example, Nahida mainly utilizes *hash* of eBPF Maps, which acts as a hash table. Benefiting from the implemented eBPF wrappers, such

as Bcc [4] and libbpf [25], we can rapidly develop eBPF programs in high-level languages. libbpf is chosen to realize Nahida for its low overhead.

3 Motivations

3.1 Existing Distributed Tracing Systems

Distributed tracing systems track end-to-end requests to help developers and operators better understand request causality and benefits for root cause localization and bottleneck troubleshooting. Distributed tracing systems are initially implemented in an invasive manner. Existing invasive tracing systems [23,30,31,35,40] require instrumenting tracing codes into programming libraries or application codes in advance so that traces can be automatically or manually generated without any user awareness. Although these existing invasive tracing systems can achieve high accuracy and enhance fault troubleshooting efficiency by code instrumentation, they still process the following distinct drawbacks.

High maintenance cost. Existing invasive tracing systems require code instrumentation to track end-to-end requests, so they have to provide libraries that implement end-to-end request tracing in multiple high-level languages for developers to use and update their instrumented libraries once the high-level programming libraries change and the same for the application source codes. The frequent updating of libraries and applications makes library maintenance time-consuming, especially when a new high-level language comes out.

Low accuracy with library instrumentation. Existing invasive tracing systems can track end-to-end requests and generate accurate traces by injecting trace contexts into messages when applications send requests to the upstream (see Figure 1). However, since applications can serve requests in multiple threads, we observed that existing invasive tracing systems with third-party library instrumentation are unable to track accurate end-to-end requests as implemented in application source codes in such complex application implementation.

Due to the above shortcomings of existing invasive systems, developers try to implement a general non-invasive distributed tracing system regardless of high-level programming languages, to reduce the maintenance costs of tracing code instrumentation. Existing non-invasive tracing systems [9,12,32,34] apply aggregation algorithms to reconstruct request causality based on the collected metadata. Although the existing algorithm-based tracing systems require zero code instrumentation, they **can not guarantee the accuracy** of the reconstructed request causality. As we observed, Deepflow [34] tracks none of the end-to-end requests under a certain workload or complex applications because of the limitations of the iterative times and policy of its aggregation algorithm (detailed in Section 6.3 and 6.4).

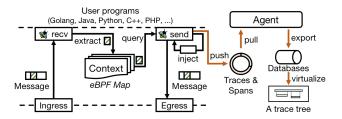


Figure 2: An overview of the workflow of Nahida.

3.2 Goals of Nahida

The popularity of distributed applications puts forward some higher requirements for distributed tracing systems based on the pros and cons of existing distributed tracing systems, which can be summarized as the following four goals.

- Goal 1: Zero instrumentation. To reduce the maintenance costs on code instrumentation, Nahida is considered to be implemented without instrumentation into applications or libraries.
- Goal 2: Support for high-level languages. Nahida should perform well on applications in modern commonly-used high-level languages, such as C, Python, Java, etc.
- Goal 3: High Accuracy. Tracking wrong request causality does harm to root cause localization when services fail in a distributed system. Nahida should keep high accuracy on tracking requests, even if applications serve requests in multiple threads.
- Goal 4: Low Overhead. Nahida should track requests at low overhead on resource consumption and service performance, without introducing extra bottlenecks.

4 System Design

4.1 Overview

Nahida employs in-kernel eBPF to generate trace context and allows to propagate the generated contexts through user applications, regardless of their implementations in different highlevel languages. To be compatible with existing distributed tracing systems, Nahida is designed to export the generated traces in the form of the popular standard, OpenTelemetry, making it possible to visualize traces in the existing user interface.

Nahida works as Figure 2 shown. Nahida first attaches eBPF programs to the system functions on message transferring, such as sendfile, sendmsg, and recvmsg. The eBPF program on the receiving side parses and extracts trace contexts from the received message, and then stores it into an eBPF map. The eBPF program on the sending side queries the trace context and injects it into the message to be sent.

If a request is finished, the eBPF programs will generate a trace with its spans and push it into a ringbuffer, an eBPF map. An agent is deployed on a host to make the generated traces accessible to users by pulling them from the ringbuffer, and exporting them to databases in OpenTelemetry standard. Users are able to access and visualize traces via their customized visualization consoles.

4.2 Trace Generation

To track end-to-end requests and generate traces, Nahida needs to generate a unique trace context for each operation in an end-to-end request, transmit the context alongside with messages, and propagate the context through user applications. More details in these procedures are shown as follows.

4.2.1 Context Generation

Inspired by OpenTelemetry and other existing distributed tracing systems, Nahida splits a request-response request into two separate operations, that is, the client requests the server and the server responds to the client. Each of the two operations can be viewed as a span in an end-to-end request. Nahida is designed to measure the time spent on these two operations of a request in the kernel. The time from the first sending of a request to the first receiving of a response is regarded as the client's time cost, while the time from the first receiving of a request to the first sending of a response is recorded as the server's time cost. For instance in Figure 1, when a client sends a request to service A, service A takes the time from its first receiving of the request to the first sending of its response as its server time, and the time from the first sending of the request to service B to the first receiving of B's response as the client time to request B. We consider the span generated when the entry service receives and serves the entry request as the root span of a trace for an end-to-end request. The other operations are all considered as children spans of that root span.

To make end-to-end requests distinguishable, Nahida assigns a unique identifier to recognize each end-to-end request, that is trace ID. Similarly, operations in an end-to-end request are distinguished by span IDs. Rules for generating trace IDs and span IDs are displayed in Figure 3. Trace ID is a 16-byte value that includes the host IP address, the timestamp when the end-to-end request entries, a sequential order, a tag for debugging, and running process ID. Span ID is 8-byte, recording the IP addresses of the communicating client and server, as well as the span order in the whole trace.

4.2.2 Context Transmission

After generating a unique trace context for each operation in an end-to-end request, Nahida has to place those contexts

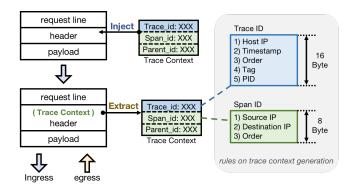


Figure 3: Rules for generating a trace ID and a span ID, when Nahida injects trace contexts into messages and extracts contexts from messages.

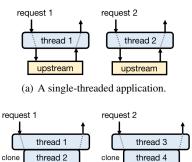
alongside requests, so that Nahida can track end-to-end requests and establish accurate request causality. The transmitting messages itself are chosen as the best way to carry trace contexts. By analyzing HyperText Transfer Protocol (HTTP) [19], its key-value header is helpful in placing our generated trace contexts. As shown in Figure 3, when sending a message, Nahida injects trace contexts as our customized key-value pairs into the header of HTTP messages before the message is sent out of the host. When receiving a message alongside with our customized trace context, Nahida intercepts the message and then extracts the context from it.

Alongside messages, trace contexts can easily travel across the network, and can be successfully transferred between two services. In the meantime, since Nahida injects context into the HTTP header instead of the payload, the injection of contexts is free from the affection on the original messages, making the injected message be accurately processed by services.

4.2.3 Context Propagation

The core of tracking an end-to-end request in a distributed tracing system is propagating the trace context through the service from one downstream request to its upstream request, to make accurate request causality(Goal 1).

The existing invasive tracing systems have implemented context propagation by instrumenting tracing codes in application source codes or programming libraries, but it is a challenge to propagate trace contexts in the kernel in a non-invasive way, without less knowledge of how a running application serves its user requests. To meet this challenge and achieve the goals mentioned in Section 3, we build up a non-invasive distributed tracing system. Based on the fact that an application is scheduled on a user process to execute by the operating system, we observe that the scheduling of an application on a user process is always changeless, where we just need to capture the user thread creation in the kernel to understand the execution of user threads as well as



(b) A simple multi-threaded applications.

upstream

upstream

Figure 4: Implementation of single-threaded and multi-threaded applications.

their parent-child relationships. Since Nahida does not care about the detailed serving process of applications, it brings Nahida a possibility to **propagate trace context with the relationships of threads** in the kernel in a non-invasive way.

Applications serve requests mainly in the following two ways.

- Single-threaded mode: Single-threaded applications serve a request in a single thread. If the serving thread has to request its upstream for services, it will send the request and wait for the response in the same thread.
- Multi-threaded mode: Multi-threaded applications always serve a request by multiple collaborating threads. When the serving thread asks its upstream for services, the application chooses to create or schedules its idle thread to request the upstream and then keeps serving the user request with no wait.

Based on these two implementations of applications, we discuss how Nahida propagates trace contexts.

For **single-threaded applications**, it is easy for Nahida to propagate trace contexts through applications with the unique identification of a thread (thread ID). Since a thread serves a request synchronously (see Figure 4(a)), Nahida can first store the extracted trace context with thread ID as key when receiving a request from downstream, and then queries that trace context for the same thread ID and injects that context into the message which will be sent to the upstream service. As a result, trace contexts can be propagated through the end-to-end requests.

For **multi-threaded applications**, they parallelize requesting the upstream service and keep serving the received user request themselves so as to improve their performance. In a multi-threaded application, once a thread serves a request from its downstream, it forks and clones a new thread to request for its upstream (see Figure 4(b)). By monitoring the

thread creation, Nahida can construct the parent-child relationships among threads. Based on these relationships, Nahida can implement context propagation. Nahida first stores the received trace context with the parent thread ID as the key. When the child thread sends a message to the upstream, Nahida finds the parent of that child thread, then figures out the trace context of that parent thread and injects it into the sending message.

4.3 Trace Collection and Visualization

After generating trace contexts and propagating them across the network and through applications, Nahida can generate a unique trace for each end-to-end request. Traces with their spans are pushed into a ringbuffer, an eBPF map, and then pulled by an agent deployed in the host. For long-term storage, the agent exports those traces to local or external databases.

Traces in the databases can be accessed and visualized in the user interface to better help developers understand and debug end-to-end requests. To be compatible with the existing visualization user interface, our agent transforms the generated traces into the standard format of OpenTelemetry before exporting them into databases.

5 Implementation

Context generation: Nahida generates 8-byte span IDs and 16-byte trace IDs according to the rules of Section 4.2.1. The time when Nahida generates a trace context is the time when a service sends a request to its upstream and a service receives a request. By parsing the content of messages, Nahida can recognize the HTTP message by matching the HTTP format and then generate trace context, without affecting other messages not in HTTP. In addition, thanks for the service governance provided by Kubernetes [6], each service and its instances have their instance IP addresses. By filtering, Nahida can further reduce the time overhead on message parsing and request tracing for non-service requests, thus more effectively serving the end-to-end request tracing for microservice systems.

Context transmission: The generated trace context needs to be injected into the sending message and extracted when received. Benefiting from the eBPF helper function, bpf_msg_push_data, Nahida can intercept the sending message and inject our customized context into it at a specific location without any user awareness. Nahida utilizes a sk_msg program to split a message into two fragments in a scatter list, then injects our customized context in the first fragment after the request line, where the first key-value data in the HTTP header is placed. The request line can be recognized by the hexadecimal ending symbol 0x0d0a of each line in an HTTP message. By doing so, Nahida can reduce the number of operating instructions required for localization finding, as

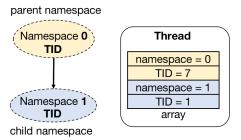


Figure 5: Thread ID assignment in Linux. Each thread ID in the child namespace has a unique ID corresponding to its parent namespace.

the eBPF program has an upper limit on the number of instructions due to its security considerations. After context injection, these two fragments are merged back as a new message to be sent. When received the request alongside the unique context, our kprobe and kretprobe programs hooked on the kernel function <code>sock_recvmsg</code> help to obtain the content of the received message and extract our injected trace context.

Context propagation: As analyzed in Section 4.2.3, to propagate trace contexts through applications, Nahida must understand parent-child relationships of threads.

In microservices, applications are deployed in containers to achieve fast deployment. However, resource isolation among containers in namespaces makes it a challenge to build up parent-child relationships of threads in the kernel. Threads of a containerized application execute in their container's namespace, and are assigned unique IDs in that namespace. In addition, threads in the children namespaces should also own unique IDs in its parent namespace. The IDs of a thread are different from namespaces, where Nahida executes in the root namespace while applications run in its children namespace. When monitoring the system function fork and clone, Nahida obtains the IDs of children threads in the children namespace and the IDs of parent threads in the root namespace, which introduces inconsistency of thread IDs and makes inaccurate parent-child relationships among threads. To meet this challenge, we try to figure out how Nahida obtains the thread IDs in different namespaces. Figure 5 shows the data structure of a thread in Linux, where a thread is regarded as a process to execute. A thread records its namespace in the variable called level and the ID of each namespace in an array named number. When a new thread is forked, it is allocated new IDs in all its located namespaces by calling the system function alloc_pid. The kprobe program on that function can help Nahida to obtain the structure of a thread and parse its IDs in all its namespace, thus Nahida is able to keep the consistency when establishing the parent-child relationships of threads.

With the collected information, to realize context propagation, Nahida stores the extracted context with thread ID as a key. When sending requests to upstreams, Nahida finds

the context to be injected. When a thread sends a message to upstreams, Nahida first looks for the trace context with running thread ID. If the running thread does not have its own trace context, Nahida continues to look for the trace context of its parent thread. After finding the trace context, Nahida will inject the child context of that trace context into the message to upstreams, thereby propagating trace contexts through applications.

Traces exportation. Our eBPF programs export traces and spans when a client receives its response and when a server sends the response. To meet the requirement for adapting to the existing tracing user interface, the agent on each host tries to read our generated traces and spans from the ringbuffer, and then export them to the external databases, with the standard of OpenTelemetry.

6 Evaluation

To evaluate the performance of Nahida, we conduct a series of experiments and mainly answer the following questions.

Q1: How Nahida work with applications implemented in multiple high-level languages?

Q2: How well Nahida can perform in the exampled microservice applications?

Q3: How stably does our tracing system perform under different concurrency?

Q4: How much is the extra overhead of Nahida on resource consumption and service performance?

6.1 Experiment Settings

6.1.1 Environment

We have set up our test environment on a Kubernetes [6] platform with 5 physical machines, each having 16GB RAM and eight 4-core 2.50GHz Intel Xeon Platinum 8269 CPUs. The machines run on Centos-8-based OS and have Linux kernel version 5.10.134. To compile the eBPF programs, we have opted for *clang* and *LLVM* version 13. We have used Fortio [21] to generate user-defined loads to our benchmarks, while the *sar* command has been utilized to obtain the CPU and memory usage of each machine.

6.1.2 Benchmark

For our experiments, we have deployed two open-source microservice application benchmarks as our testbed. The first benchmark is Bookinfo [7], which is an online bookstore serving in microservices. Bookinfo is written in various programming languages, including Python, and Java. The second benchmark is Trainticket [22], which is a large-scale distributed train ticket booking system consisting of 41 microservices, written in Java. Each service of these two benchmarks

owns one replica. Both Bookinfo and Trainticket are singlethreaded, and thus they are regarded as our benchmarks on single-threaded applications. Since these two benchmarks do not provide multi-threaded examples, we refactored the source code of the frontend service of Bookinfo and Trainticket to build up multi-threaded applications, which can be regarded as our benchmarks on multi-threaded applications.

In addition, we conducted both single-threaded and multithreaded benchmarks in multiple high-level languages, including PHP, Java, Python, and Golang to test how Nahida performs on them. Each example benchmark consists of three services, where the frontend service asks the middle service for content and the middle requests the backend.

6.1.3 Baselines

Jaeger with OpenTracing [23, 31] is a commonly used approach for tracking end-to-end requests in a distributed system. Jaeger stands for a typical invasive implementation of a distributed tracing system, which adheres to the OpenTelemetry/OpenTracing standard and instruments tracing codes directly into service source codes and can be considered as the most accurate baseline. Deepflow [34] is state-of-the-art non-invasive approaches for tracking requests, which implements an aggregation algorithm based on its collected metadata. We compare Nahida with Deepflow to show our performance.

6.1.4 Evaluation Metrics

To better evaluate the effectiveness of Nahida, we use the precision as our metric. Precision reflects the similarity between the request traces generated by different tracing systems and the actual request invocation relationship. The higher the precision, the more effective the tracing system. To measure the impact of Nahida on service performance, we use two metrics: request latency and query-per-second (QPS). Request latency is used to show the impact of Nahida on service response time. QPS is used to measure the throughput of the service. In addition, we also measure the consumption of CPU and memory to help understand the resource consumption of Nahida on the hosts. By using these metrics, we can analyze the overhead of Nahida.

6.2 (Q1) Multi-language Supporting

To test the performance of Nahida on supporting multilanguage application tracing, we used Fortio to generate a total of ten thousand requests continuously and sequentially for our constructed single-threaded and multi-threaded benchmarks in multiple high-level languages. We collect the evaluation metrics and draw Figure 6, where "S" represents the service responses in single-threaded mode, while "M" represents replying in multi-threaded mode. We test PHP just in single-threaded mode due to its widespread use.

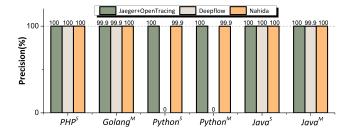


Figure 6: Effectiveness comparison between Nahida, Deepflow, and Jaeger with OpenTracing when they track requests served by multiple high-level languages.

As shown in Figure 6, Jaeger can always achieve perfect performance on tracing end-to-end requests by instrumenting tracing codes into applications in multiple high-level languages. Deepflow also supports multi-language applications. With the monitored data on network communication, Deepflow analyzes causality between requests, regardless of the implementation of applications. As we can see, Deepflow can aggregate almost all request causality for PHP, Golang, and Java applications, but it does bad in tracking end-to-end requests for Python applications, no matter if the application is implemented in the single-threaded mode or multi-threaded mode. By recognizing the trace log collected from Deepflow, we find out that requests can not be correctly associated with the value of a tag syscall_trace_id, which is often zero, resulting in a failure to restore the complete end-to-end trace.

Nahida can track nearly 100% of end-to-end requests of multi-language applications, even those applications serve requests in multi-threaded mode. Monitoring parent-child relationships of threads, Nahida can recognize in the kernel which thread that a request belongs to, thus propagating trace contexts through applications from one thread to another thread, without detailed invasive information of applications. The result shows that Nahida performs well in tracking the end-to-end requests in a non-invasive way.

6.3 (Q2) Microservices Examples

To evaluate the performance of Nahida on tracking end-to-end requests of containerized microservices, we choose Bookinfo and Trainticket as our benchmarks. Fortio is set to generate a total of ten thousand requests continuously and sequentially for the single-threaded and multi-threaded Bookinfo and Trainticket. The comparison result on accuracy is shown in figure 7.

Jaeger still keeps tracking end-to-end requests perfectly with instrumentation into application codes. Deepflow fails to reconstruct accurate causality between requests, since the tags named <code>syscall_trace_id</code> of trace logs are always zero which shows no causality relationships among requests and makes a parent request lose its child request and thus fails to generate an end-to-end trace. Meanwhile, the aggregation

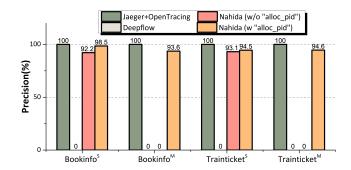


Figure 7: Effectiveness comparison between Nahida, Deepflow, and Jaeger with OpenTracing when they track requests for single-threaded and multi-threaded Bookinfo and Trainticket.

algorithm of Deepflow has a limit of the iterative times. When the number of service calls involved in an end-to-end request is too many, all the involved service calls cannot be traversed under the limited iterative times. For example, for Travel service requests with 97 spans under Trainticket, Deepflow could not restore the end-to-end requests.

Nahida performs better than Deepflow even tracking end-toend requests of containerized microservices. Since a container works as a process in the host, Nahida can easily find the parent-child relationship among its threads and propagate trace context through the application, thus achieving accuracy on tracking requests of at least 93.6%, even tracking for a total of 97 spans for an end-to-end request to Trainticket. The inaccurate and incomplete traces are generated when the Detail service or the Reviews service does not generate their spans normally. By observing the logs from eBPF, we find that after the service receives a request and records its trace context, the eBPF program is not triggered when sending the response, resulting in no span generated on the server side, thus affecting the accuracy of the trace generation.

From Figure 7, We can observe that the eBPF program hooked on function alloc_pid helps Nahida well to build a parent-child relationship for containerized service threads on the host, regardless of the resource isolation of namespaces. Without monitoring the function alloc_pid, Nahida tracks end-to-end requests well of the single-threaded Bookinfo and Trainticket application, since a thread processes a request individually. However, because of the resource isolation of namespaces, Nahida can not find the parent-child relationships between two threads in different namespaces, so Nahida monitoring the function sys_clone instead of alloc_pid fails to track end-to-end requests of multi-threaded Bookinfo and Trainticket.

6.4 (Q3) Concurrency Impact

To evaluate the performance of Nahida under different concurrency, we selected Bookinfo as our benchmark since it

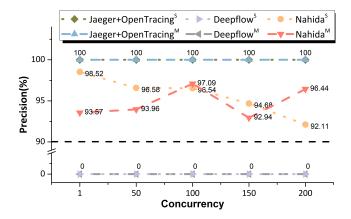


Figure 8: Effectiveness comparison between Nahida, Deepflow, and Jaeger with OpenTracing when they track concurrent requests for single-threaded Bookinfo.

can process up to two hundred concurrent requests simultaneously, whereas Trainticket can only process less than one hundred requests at the same time. Fortio is configured to generate a total of ten thousand requests for both single-threaded and multi-threaded Bookinfo, with 1, 50, 100, 150, and 200 concurrent users, respectively. Compared results with Jaeger and Deepflow are presented in Figure 8.

Jaeger keeps tracking all the end-to-end requests of either single-threaded or multi-threaded applications under different concurrent user workloads. With explicit instrumentation in application source codes, Jaeger can easily distinguish which downstream request caused each concurrent upstream request, thus tracking the exact causal relationship between requests.

In the case of concurrent user requests, Deepflow can aggregate part of the Bookinfo application's end-to-end requests, but it cannot support aggregating the complete end-to-end requests with a metadata-based algorithm, so the accuracy of its reconstructed request causality is almost zero. For example, the Productpage service needs to request the Detail service and the Reviews service, respectively, to make a response to a user request. Deepflow treats these two requests as two separate traces rather than a trace for an end-to-end request. Observed from trace logs collected by Deepflow, Deepflow utilizes a tag named syscall_trace_id to find out the causality relationships between two trace logs, but it fails to assign accurate syscall_trace_id to associate the request from Productpage to Details with the request from Productpage to Reviews, which are likely to be zero. In addition, Deepflow applies the policy that the multiple requests or responses in the same transmission direction share the same trace ID, so that it confuses the causal relationship between many requests.

Nahida can achieve accuracy at more than 92% when tracking concurrent end-to-end requests for single-threaded and multi-threaded applications. By propagating trace context with parent-child relationships of threads, Nahida can per-

Table 1: Overhead Comparison between the normal (Nahida-) and Nahida (Nahida+)

Metric	Concurrency	Nahida-	Nahida+	Overhead (Δ)
Latency	100	683.85	694.67	10.82 (1.58%)
	200	1374.76	1403.73	28.97 (2.1%)
QPS	100	146.17	143.9	2.27 (1.55%)
	200	145.33	142.33	3.0 (2.06%)
CPU	100	10.65	10.76	0.11 (1.0%)
	200	10.49	10.83	0.34 (3.3%)
Memory	100	72.27	74.32	2.05 (2.84%)
	200	72.12	74.84	2.73 (3.78%)

form well on request tracking, stably. The reason why Nahida can not achieve 100% accuracy is that the eBPF program hooked on the sending and receiving system functions can not perform properly due to high concurrency. As a result, Nahida fails to generate some of the spans.

6.5 (Q4) Overhead

We generated workloads with 100 and 200 concurrent users using Fortio to evaluate the overhead of Nahida when tracking requests for the single-threaded Bookinfo. The experiments run for three minutes and were repeated three times on five physical machines. The average results are presented in Table 1.

First, we examine the impact of Nahida on the service performance. As shown in Table 1, Nahida increases the request latency by about 10.82–28.97 ms (1.58%–2.1%) compared to the original, and reduces the QPS of the service by about 2.27–3.0 (1.55%–2.06%), even under high concurrency, such as two hundred. The additional latency comes from our customized eBPF programs when Nahida intercepts and parses the sending and receiving messages in the Linux kernel.

Regarding resource consumption, Nahida has increased the average CPU usage of machines by 0.11%–0.34% (1.0%–3.3% relative to the original). Our eBPF programs executing in the Linux kernel cause extra CPU consumption. In the meantime, Nahida needs 2.05%–2.73% (about 336MB–447MB) more of the memory to temporarily store part of the messages, the generated trace context and other information.

As demonstrated in Table 1, Nahida has a stable impact on service performance and resource consumption, regardless of the number of concurrent requests served by the service. Overall, compared to the original, Nahida has a 1.55%–2.1% loss in service performance and requires an additional 1.0%–3.78% CPU and memory consumption.

6.6 Discussion

We implement a non-invasive distributed tracing system, Nahida, for applications serving the HTTP message. There exist some limitations in our implementation, which lead to our future work. First, messages in protocols other than HTTP, such as HTTPS and gRPC, are encrypted, which makes it difficult for Nahida to inject trace contexts into those messages in the kernel, due to their unstructured data. In this case, we can inject the trace context at the front of a request instead of in the header to propagate trace contexts between services in our future work. This approach would require the receiver to deploy an in-kernel eBPF program to resolve and remove the context from the message without affecting the upstream service serving request messages. Second, eBPF programs can only be deployed by privileged users. But Linux has the plan to make them available to unprivileged users [14], and there exists some work on unleashing unprivileged eBPF programs [26]. The requirement to root privilege of eBPF programs may be alleviated in the future. Third, the mounting of eBPF programs to some user functions requires ZeroTracer to obtain the addresses of the user functions in different thirdparty libraries. Fourth, ZeroTracer requires a Linux kernel version of at least 5.2 [5] to execute. Fifth, ZeroTracer is compatible with the existing OpenTelemetry and does not affect request tracing of OpenTelemetry.

7 Related Work

Invasive tracing systems require developers to instrument tracing codes into their service codes to generate traces. Zipkin [40], Jaeger [23], SkyWalking [2], OpenTracing [31], OpenTelemetry [30], and Dapper [35] are some typical examples of invasive tracing systems. In these tracing systems, applications or programming libraries must be instrumented either in the requester's codes or in the requestee's codes. Castanheira et al. [8] proposed a flexible distributed tracing workflow based on instrumented annotations and preemptive tracing for debugging. JCallGraph [27] realizes a tracing system for microservice applications by instrumenting the cloud platform middlewares. However, as applications and libraries are updated frequently, developers have to repeatedly instrument the updated codes, which consumes significant time. Instrumenting the programming libraries can achieve user transparency, but applications need to be executed in the instrumented environment. On the other hand, instrumented applications can be executed in any environment, but require explicit instrumentation by the developer.

Existing non-invasive tracing systems generate traces of requests without any instrumentation into applications and programming libraries. Inkle [32] tracks gRPC requests for microservice applications on Kubernetes. Cha *et al.* [9] constructed the request traces based on the information collected by the service proxies in the service mesh. Both approaches

can only generate single-hop traces without instrumenting the source codes. DeepFlow [34], and ContainIQ [12] mount eBPF programs at some of the Linux system calls and user-defined functions to collect commonly-used metrics and generate traces by an aggregation algorithm or metadata-based algorithm. However, both approaches may suffer from a loss of accuracy.

8 Conclusion

To track requests in a large-scale distributed system in a non-invasive way, we propose Nahida, based on eBPF, regardless of high-level programming language and application implementations. Our evaluations demonstrate that Nahida can track completely over 92% of requests, maintaining stable tracing performance with increased concurrency. Notably, Nahida can fully track requests served by multi-threaded applications, a capability that existing invasive tracing systems that instrument tracing codes in programming libraries lack. Furthermore, our tests indicate that Nahida has little impact on service performance, with only a 1.55%–2.1% increase in service latency. In terms of resource consumption, Nahida consumes an additional 1.0%–3.78% of CPU and memory resources, compared to the normal system.

References

- [1] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. Inkev: In-kernel distributed network virtualization for dcn. *SIGCOMM Comput. Commun. Rev.*, 46(3):1–6, 2018.
- [2] Apache. Skywalking: Application performance monitor tool for distributed systems, especially designed for microservices, cloud native and container-based (kubernetes) architectures. https://skywalking.apache.org/. Accessed March 1, 2023.
- [3] Sabur Baidya, Yan Chen, and Marco Levorato. ebpf-based content and computation-aware communication for real-time edge computing. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 865–870. IEEE, 2018.
- [4] Bcc. Bcc: Tools for bpf-based linux io analysis, networking, monitoring, and more. https://github.com/iovisor/bcc. Accessed February 6, 2023.
- [5] Bcc. Bpf features by linux kernel version. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md, 2023. Accessed September 6, 2023.

- [6] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81– 84, 2014.
- [7] Bookinfo. Bookinfo. https://istio.io/latest/docs/examples/bookinfo/. Accessed February 6, 2023.
- [8] Lucas Castanheira, Theophilus A. Benson, and Alberto Schaeffer-Filho. The case for more flexible distributed tracing. In *Proceedings of the Student Workshop*, CoNEXT'20, page 27–28. Association for Computing Machinery, 2020.
- [9] Donghun Cha and Younghan Kim. Service mesh based distributed tracing system. In 2021 International Conference on Information and Communication Technology Convergence (ICTC), pages 1464–1466, 2021.
- [10] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings Interna*tional Conference on Dependable Systems and Networks, pages 595–604, 2002.
- [11] YoungEun Choe, Jun-Sik Shin, Seunghyung Lee, and JongWon Kim. ebpf/xdp based network traffic visualization and dos mitigation for intelligent service protection. In *International Conference on Emerging Internetwork*ing, Data & Web Technologies, pages 458–468. Springer, 2020.
- [12] ContainIQ. Containiq. https://www.containiq.com/. Accessed March 2, 2023.
- [13] Jonathan Corbet. Extending extended bpf. *Linux Weekly News*, 2014.
- [14] Jonathan Corbet. Unprivileged bpf(). https://lwn.net/Articles/660331/, 2015. Accessed September 6, 2023.
- [15] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.
- [16] eBPF. ebpf. https://ebpf.io, 2022. Accessed February 6, 2023.
- [17] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.*, 30(4), 2012.
- [18] Facebook. Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran. Accessed February 6, 2023.

- [19] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol-http/1.1. Technical report, 1999.
- [20] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-Trace: A pervasive network tracing framework. In 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07). USENIX Association, April 2007.
- [21] Fortio. Fortio load testing library, command line tool, advanced echo server and web ui in go (golang). https://github.com/fortio/fortio. Accessed February 6, 2023.
- [22] FudanSELab. Train ticket a benchmark microservice system. https://github.com/FudanSELab/train-ticket. Accessed February 6, 2023.
- [23] Jaeger. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/. Accessed March 1, 2023.
- [24] Simon Jouet and Dimitrios P Pezaros. Bpfabric: Data plane programmability for software defined networks. In 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pages 38–48. IEEE, 2017.
- [25] libbpf. libbpf: a userspace library for loading and interacting with bpf programs. https://github.com/libbpf/libbpf, 2020. Accessed February 6, 2023.
- [26] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing unprivileged ebpf potential with dynamic sand-boxing. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 42–48, New York, NY, USA, 2023. Association for Computing Machinery.
- [27] Haifeng Liu, Jinjun Zhang, Huasong Shan, Min Li, Yuan Chen, Xiaofeng He, and Xiaowei Li. Jcallgraph: Tracing microservices in very large scale container cloud platforms. In Cloud Computing – CLOUD 2019: 12th International Conference, Held as Part of the Services Conference Federation, SCF 2019, San Diego, CA, USA, June 25–30, 2019, Proceedings, page 287–302. Springer-Verlag, 2019.
- [28] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings, USENIX'93, page 2. USENIX Association, 1993.
- [29] Sape Mullender. Distributed systems. ACM, 1990.

- [30] OpenTelemetry. Opentelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability. https://opentelemetry.io/. Accessed March 1, 2023.
- [31] OpenTracing. Opentracing: Vendor-neutral apis and instrumentation for distributed tracing. https://opentracing.io/. Accessed March 1, 2023.
- [32] Abram Perdanaputra and Achmad Imam Kistijantoro. Transparent tracing system on grpc based microservice applications running on kubernetes. In 2020 7th International Conference on Advance Informatics: Concepts, Theory and Applications (ICAICTA), pages 1–5, 2020.
- [33] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, page 780–794. Association for Computing Machinery, 2022.
- [34] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, page 420–437. Association for Computing Machinery, 2023.
- [35] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.
- [36] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.
- [37] Maarten Van Steen and Andrew S Tanenbaum. A brief introduction to distributed systems. *Computing*, 98:967– 1009, 2016.
- [38] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In Mobile Object Systems Towards the Programmable Internet: Second International Workshop, MOS'96 Linz, Austria, July 8–9, 1996 Selected Presentations and Invited Papers, pages 49–64. Springer, 2005.
- [39] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. Leveraging ebpf for programmable network functions with ipv6 segment routing. In *Proceed*ings of the 14th International Conference on emerging Networking Experiments and Technologies, pages 67– 72. Association for Computing Machinery, 2018.

[40] Zipkin. Zipkin is a distributed tracing system. https://zipkin.io/. Accessed March 1, 2023.