Interoperability From OpenTelemetry to Kieker: Demonstrated as Export from the Astronomy Shop

David Georg Reichelt
Lancaster University Leipzig / URZ Leipzig

Shinhyung Yang
Kiel University

Wilhelm Hasselbring
Kiel University

Abstract

The observability framework Kieker provides a range of analysis capabilities, but it is currently only able to instrument a smaller selection of languages and technologies, including Java, C, Fortran, and Python. The OpenTelemetry standard aims for providing reference implementations for most programming languages, including C# and JavaScript, that are currently not supported by Kieker. In this work, we describe how to transform OpenTelemetry tracing data into the Kieker framework. Thereby, it becomes possible to create for example call trees from OpenTelemetry instrumentations. We demonstrate the usability of our approach by visualizing trace data of the Astronomy Shop, which is an OpenTelemetry demo application.

1 Introduction

To understand the behavior of a software system, observability tools are used. Different observability tools provide different capabilities: While Kieker [9] has various analysis capabilities and is known for its low overhead [3], OpenTelemetry is the de-facto standard for obtaining data and standard implementations provide agents for a variety of languages [5]. To make them interoperable, three steps are necessary: (1) Transformation of Kieker traces into the OpenTelemetry format, (2) Transformation of OpenTelemetry traces into the Kieker format, and (3) Usage of native OpenTelemetry data formats in Kieker. The first step has been done in our prior work [7].

In this work, we present the implementation of the second step. By transformation of OpenTelemetry traces into Kieker,[1] we make it possible to use OpenTelemetries rich agent landscape within the Kieker analysis framework. Thereby, it becomes also feasible to do a range of Kieker analysis. Our analysis shows a structural difference between Kieker and OpenTelemetry: While Kieker traces are synchronous traces, supporting the observation of one control flow through a program, OpenTelemetry aims at representing asynchronous traces, supporting the observation of microservice calls. This results in different data structures and therefore limited compatibility of the concepts.

[1] Traces are recevied via gRPC, e.g., from an agent.

The remainder of the paper is structured as follows: First, we introduce the data formats of Kieker and OpenTelemetry. Based on this, we describe possible approaches for conversion of the data formats. Subsequently, we demonstrate the application of the conversion within the OpenTelemetry demo. Afterwards, we compare our approach to related work. Finally, we give a summary and an outlook.

2 Data Formats

In this section, we describe Kieker's and OpenTelemetry's data formats.

Kieker Within Kieker's monitoring part, the so-called monitoring log[2] is obtained. This monitoring log is stored somewhere persistently to allow further analysis. These analyses consist of a variety of stages, that first read the monitoring data, transform them into traces and then provide analysis results, e.g., call trees and component graphs.

This architecture relies heavily on the data formats allowed within the monitoring log. These data formats are created using the Instrumentation Record Languages (IRL).[3] Based on an Xtext definition of records, the IRL allows to create source code for the usage of these records within the Kieker monitoring and analysis components.

OpenTelemetry OpenTelemetry provides data format standards for the three pillars of observability: Logs, metrics, and traces. Logs are timestamped text records, metrics are quantitive measurements of a system, and traces are a sequence of events which make it possible to follow an execution path. All OpenTelemetry data are serialized with protobuf serialization, ensuring high compression and low overhead for serialization and deserialization.[4]

For observability, we consider **traces** the most important pillar. A trace consists of spans, which itself contains a name, a start and an end timestamp, a list of fields, a list of attributes and a list of

[2] In OpenTelemetry's terminology, this contains both tracing and monitoring information.

[3] `https://github.com/kieker-monitoring/instrumentation-languages/wiki`

[4] `https://github.com/open-telemetry/opentelemetry-proto`

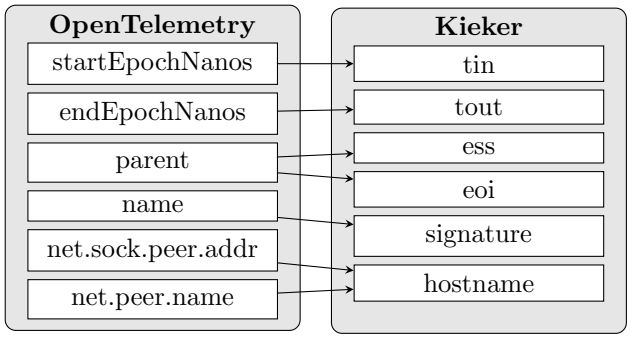

Figure 1: Mapping of Fields

events. The attributes are arbitrary key/value mappings, e.g., `net.sock.peer.addr` might be mapped to `127.0.0.1`. By the trace context level W3C recommendation,[5] that is implemented by OpenTelemetry, span contexts can be obtained across services.

Besides standard information like start and end time, spans can also contain attributes. These attributes are key-value pairs. The OpenTelemetry semantic conventions provide guidelines on how these attributes should be used.[6]

3 Data Conversion

For the basic data, most OpenTelemetry fields can just be mapped to their counterparts in Kieker. For the control flow, a more sophisticated approach is necessary.

Mappings For converting the data, most conversions are just simple renamings: The time stamps and the signature can just be renamed. The spans name is not necessarily a method call, it can also be for example an HTTP call; however, using the name as signature is the best mapping that can be used here. Furthermore, the hostname can be created from a combination of attributes defined in the OpenTelemetry semantic conventions. Figure 1 represents the mapping of fields.

Control Flow For the control flow, OpenTelemetry and Kieker contained not fully compatible concepts: OpenTelemetry represents asynchronous traces, where every span is part of a parent span. One parent span might have multiple child spans taking place at the same time, and the child span that has been started first might end after the child span that has been started second.

In contrast, Kieker represents synchronous traces, where only one call can take place at the same time. If parallel processing happens, this is represented by separate traces. Internally, Kieker stores an execution order index (`eoi`) and the execution stack size (`ess`) of each invocation, which makes it fast to persist the current tracing situation within the tracing agent. The `ess` is only allowed to increase by 1, which happens if a method calls a child method.

While OpenTelemetry represents the control flow using a `parent` reference, Kieker stores the `ess` and `eoi` of each invocation. OpenTelemetry's implementation thereby supports asynchronous calls within their traces, Kieker's implementation does not make it necessary to have references in serialized data, which reduces the overhead during tracing.

Since Kieker assumes that traces are sequential, it checks whether parents can be assigned correctly. This process breaks in situations like the one depicted in Figure 2: Here, a root method calls `call1` and `call2` asynchronously. `call1` returns, but `call2` still goes on. Later, the root span calls `call3`, and `call2` calls `call4`. The spans are ordered by their starting timestamp. Kieker would usually try to assign `call4` to a parent, but since `call3` already started, there is a gap in the `ess` by two, which is a sign of an inconsistent trace.

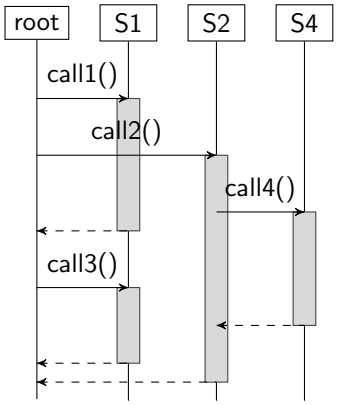

Figure 2: Unrepresentable Parallel Trace

To overcome this problems, there are four main solutions: (1) Linearize the traces: The calls could be located next to their caller, regardless of when they happen. While this would remove the issue, it would require storing all spans as long as there could be a potential child span arriving, which would increase the resource usage of the trace receiver heavily. (2) Directly convert the traces: Instead of storing traces as Kieker records into the monitoring log and loading them, OpenTelemetry traces could be transformed into Kieker's `ExecutionTrace` directly. This would break the basic split of Kieker's monitoring and analysis components. (3) Create an additional record that represents asynchronous spans, and can be persisted, loaded and analysed with separate Kieker implementations. (4) Mark traces as asynchronous: Mark a trace as asynchronous during execution and change the parent assignment if this flag is activated.

Solution (1) is unacceptable due to the resource usage and solution (2) is unacceptable since it would make repeated analyses impossible. Solution (3) is possible, but would require rewriting big parts of Kieker's analysis pipelines. Therefore, for now, we decied to go with option (4). This requires specifying `--asynchronousTrace` to `kieker-trace-analysis` when an OpenTelemetry trace is loaded into it.

This flag is necessary to process all asynchronous traces within the `kieker-trace-analysis`. For

[5] `https://www.w3.org/TR/trace-context-2/`

[6] `https://opentelemetry.io/docs/specs/semconv/general/attributes/`

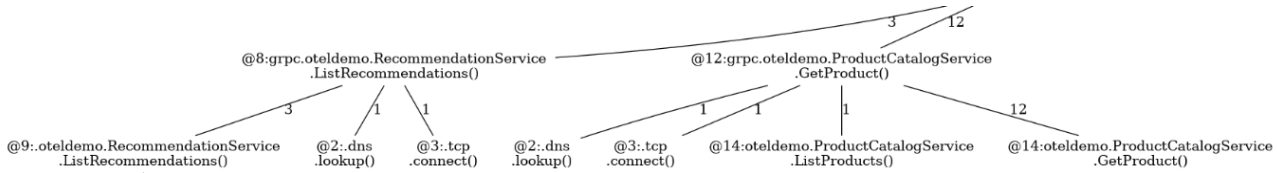

Figure 3: Part of Call Tree

strictly synchronous traces, e.g., the traces from MooBench, this is not necessary. For traces from microservice applications like the TeaStore [1], the Astronomy shop or the T2 Shop [4], it will always be necessary to set the `--asynchronousTrace` flag.

4 Case Study: Visualizing Astronomy Shop Traces

The astronomy shop is the default demo application for OpenTelemetry-related tools.[7] It consists of 14 services written in 11 different languages, which makes it challenging to instrument them and to establish connections among the agents. Kieker is not able to instrument .NET and TypeScript, therefore, it could not be used to instrument the astronomy shop; however, OpenTelemetry's instrumentation is able to do so. To showcase our data transformation, we started the astronomy shop with instrumentation and activated our Kieker-otel-transformer. Afterwards, we visualized the obtained trace data using Kieker's trace analysis tool. Figure 3 shows a part of the created call tree, which visualized the calls from two different services, the product and the recommendation service.

5 Related Work

Various works exist that examine the transformation of OpenTelemetry traces. Weber et al. [8] examine the interoperability of OpenTelemetry and Palladio. In order to continuously predict the performance, they generate Palladio models based on OpenTelemetry. TraceZip is an approach for storing OpenTelemetry data in a compressed format [6]. Thereby, they are able to reduce the memory requirement within a TrainTicket-based benchmark by up to 33.8 %. Additionally, the throughput is increased, at the cost of increased CPU utilization due to compression. These works process OpenTelemetry traces, but none of them is able to execute the Kieker analysis with OpenTelemetry traces. Additionaly, there is research on model transformation in general. Groner et al. [2] research the view of developers on data model transformations. They find that more than half of the participants already tried to improve the performance of their transformations. While we focused on implementing a prototype in this work, examining the performance of different implementations of the transformation would be valuable future work.

[7] `https://github.com/open-telemetry/opentelemetry-demo`

6 Summary and Outlook

In this work, we discussed how to achieve the ability to use OpenTelemetry data in the Kieker analysis pipeline. The main issue is the different purpose of the data formats: While Kieker represents sequential traces, OpenTelemetry represents asynchronous traces. We overcome this by marking traces as asynchronous. Thereby, we made it possible to analyse the OpenTelemetry tracing data from the Astronomy Shop using Kieker. This was the second step of our efforts to make Kieker and OpenTelemetry interoperable. The final step is to support the OpenTelemetry data format fully within Kieker, which requires rewriting parts of the analysis pipeline. Another possible work could be the combination of traces: To have low overhead, applications could be traced inside with Kieker and outside of the application, OpenTelemetry could be used to trace the service calls. By this approach, a combination of Kieker's low-overhead and OpenTelemetry's widespread applicability could be achieved.

References

[1] S. Eismann et al. "TeaStore: A Micro-Service Reference Application for Cloud Researchers". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*. IEEE, Dec. 2018. DOI: 10.1109/ucc-companion.2018.00021.

[2] R. Groner et al. "An exploratory study on performance engineering in model transformations". In: *ICMDELS*. 2020, pp. 308–319. DOI: 10.1145/3365438.3410950.

[3] D. G. Reichelt, S. Kühne, and W. Hasselbring. "Overhead Comparison of OpenTelemetry, inspectIT and Kieker". In: *SSP 2021*. 2021. URL: https://ceur-ws.org/Vol-3043/.

[4] S. Speth, S. Stieß, and S. Becker. "A saga pattern microservice reference architecture for an elastic SLO violation analysis". In: *2022 IEEE 19th ICSA-C*. IEEE. 2022, pp. 116–119.

[5] D. G. Blanco. *Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization*. APress, 2023. DOI: 10.1007/978-1-4842-9075-0.

[6] Z. Chen, J. Pu, and Z. Zheng. "Tracezip: Efficient Distributed Tracing via Trace Compression". In: *ISSTA* (2025), pp. 411–433. DOI: 10.1145/3728888.

[7] D. G. Reichelt et al. "Interoperability From Kieker to OpenTelemetry: Demonstrated as Export to ExplorViz". In: *SSP 2024*. PID: 20.500.12116/46200. 2025, pp. 20–22.

[8] S. Weber, T. Weber, and J. Henß. "Integration of performability-model extraction and performability prediction in continuous integration/continuous delivery". In: *SSP 2024*. PID: 20.500.12116/46177. 2025, pp. 29–31.

[9] S. Yang et al. "The Kieker Observability Framework Version 2". In: *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*. 2025, pp. 11–15. DOI: 10.1145/3680256.3721972.