2023

Tracing & Observability

RESEARCH DOCUMENT

KOEN EVERS

Table of contents

Glossary

1. Observability: The extent to which a system can be monitored and analyzed, providing insight into its behavior and performance. It involves collecting and analyzing metrics, logs, and traces to gain a comprehensive understanding of the system.
2. OpenTelemetry: A library for instrumentation that aims to provide standardized, vendor-agnostic SDKs, APIs, and tooling for collecting, transforming, and forwarding telemetry data to an observability back-end. It offers flexibility in integration and implementation within code.
3. Trace: A distributed trace records the paths taken by requests or operations as they propagate through multi-service architectures, such as microservice and serverless applications. Traces help pinpoint the cause of performance problems in distributed systems.
4. Span: Spans are the building blocks of a trace. They represent individual units of work or operations within a trace and provide information such as the duration of a function or operation.
5. Back-end: The back-end refers to the infrastructure or system where the output of traces is stored and visualized.
6. Zipkin: A tracing back-end written in Java, it uses a centralized collector model for deployment. It supports multiple languages and has official support for popular libraries, with community support for less popular ones.
7. Jaeger: Another tracing back-end written in Go, it uses a decentralized collector model for deployment, making it easier to scale and deploy via Kubernetes. It supports most languages and has extensive support for libraries, particularly OpenTracing. Jaeger also integrates well with cloud platforms and has out-of-the-box support for gRPC.
8. Instrumentation: The process of modifying code or adding additional code to track and collect telemetry data, such as traces and spans. Instrumentation enables capturing the behaviour and performance of a system.
9. STDOUT: Standard output, a stream where text-based output is typically displayed or written to in command-line interfaces. In the context of tracing, it refers to the option of outputting traces to the STDOUT exporter.
10. gRPC: A high-performance, open-source framework for remote procedure calls (RPC) that enables communication between different services or components in a distributed system.
11. ElasticSearch: A popular open-source search engine and analytics platform used for storing and indexing large amounts of data. It is mentioned as a choice for the datastore component in the implementation of the tracing solution.
12. SDK (Software Development Kit): A collection of software tools and libraries that developers use to create applications for a specific platform or software framework. It provides pre-built functions, APIs, and utilities to simplify development tasks.

Research Context.

This research is meant to help answer the following primary research question:

How can tracing help improve the Observability of a software product so that bottlenecks can be identified.

This is an important topic this semester as we are focused on micro services. Small independently running products that are able to scale up and down as the demand on them increases or decreases. It is therefore important to keep performance in mind. Bottlenecks can cause micro services to scale up and thus increase hosting costs. Having good observability of the way your code and services work, being able to see where code takes long to run and where it runs fast lets you focus your attention to the necessary parts of your code. Tracing is a way for a programmer to instrument their code so that the flow and speed of their code can be measured and even logged. Because of these reasons I believe the topic of Observability to be a valuable research topic.

But what is observability?

Observability.

What is observability when we talk about code? To answer this, I consulted colleagues and an online resource (What is Observability? A Beginner's Guide, z.d.).

Observability is the extent to which a system can be monitored and analysed, providing insight into a system's behaviour. Metrics, logs and traces are all types of data that can be collected and used to gain insight into a system's behaviour and performance.

Metrics can be used to monitor system performance over time and identify any trends or patterns that may indicate a problem. Logs can be used to understand events within the system and detect errors or exceptions. Traces can be used to understand the flow of a system and identify any bottlenecks or other performance issues.

By collecting and analysing these types of data, it is possible to gain a more complete understanding of a system's behaviour and to identify and solve any problems. In short, Observability is often enhanced by collecting and analysing metrics, logs and traces.

Traces

When we talk about observability in the context of systems or software, traces are an essential component. Traces provide a detailed record of the execution path and behavior of a system or application.

In software systems, a trace refers to a sequence of events or log entries that capture information about the interactions between different components or services. These events could include method calls, network requests, database queries, and other relevant activities that occur during the execution of the system.

Traces are typically used to gain insights into the performance, behavior, and dependencies of a system. By collecting and analyzing traces, you can identify bottlenecks, diagnose issues, and understand how different parts of the system are interacting with each other.

Traces often include additional metadata such as timestamps, unique identifiers, and contextual information, which help in correlating events and reconstructing the overall flow of execution. This allows developers and system administrators to troubleshoot problems, optimize performance, and ensure the reliability of the system.

Overall, traces are a valuable tool for understanding the inner workings of complex systems, enabling better observability and the ability to analyze and improve system behavior.

Research Setup.

The setup of this document is as follows:
I will attempt to answer the primary research question through the use of the following sub questions whilst utilising the listed DOT-research methods.

The sub questions:

1. Which tracing solutions exist?
    a. Available product analysis
    b. Literature study
2. How to implement a tracing solution?
    a. Literature study
    b. Document Analysis
    c. Prototyping
3. Can a tracing solution help detect bottlenecks
    a. Literature study
    b. Document analysis
    c. Data analysis
    d. Prototyping
    e. Peer Review

I hope these questions will bring me to a useful conclusion to the primary research question.

1. Which tracing solutions exist?

There is really only one way to implement tracing these days. Cossack Labs (How to Implement Tracing in a Modern Distributed Application, 2018) tells us why. A few years ago, there were multiple options with OpenTracing and OpenCensus but these two systems moved forward together as OpenTelemetry on May 2019. With this, OpenTelemetry has actually become the industry standard where there is choice in how you store and display the data. Although recently there has been improved support for tracing by for example .NET app, where support started midway through 2022 and is still being updated in 2023. .NET recommends either OpenTelemetry or Application Insights.

Application Insights is a Microsoft specific distributed tracing solution which has support for tools such as Azure Monitor but also for the following languages: .Net, .Net Core, Java, Node.js, JavaScript and Python.

What is OpenTelemetry?

Due to the more limited nature of Application Insights I decided to continue my research primarily focusing on OpenTelemetry. So what is OpenTelemetry?

OpenTelemetry aims to provide standardised, vendor agnostic SDKs, APIs and tooling that can be used to ingest, transform and forward data to an Observability back-end. According to OpenTelemetry (Observability Primer, 2022) itself, OpenTelemetry provides you with the following:

- *"A single, vendor-agnostic instrumentation library per language with support for both automatic and manual instrumentation.*
- *A single vendor-neutral collector binary that can be deployed in a variety of ways.*
- *An end-to-end implementation to generate, emit, collect, process, and export telemetry data.*
- *Full control of your data with the ability to send data to multiple destinations in parallel through configuration.*
- *Open-standard semantic conventions to ensure vendor-agnostic data collection.*
- *The ability to support multiple context propagation formats in parallel to assist with migrating as standards evolve.*
- *A path forward no matter where you are on your observability journey."*

OpenTelemetry is thus a library for instrumentation that is very flexible in terms of integration options and simple to deploy. Furthermore, OpenTelemetry also gives a lot of freedom in how you implement it within the code. You can implement OpenTelemetry quite easily according to OpenTelemetry's standard components or you can modify the components yourself for your own custom solution. This flexibility makes OpenTelemetry a great choice to implement for observability as it can fit a wide range of technologies and environments.

Trace & Span.

So OpenTelemetry is a library for instrumentation using Traces, among other things. but what is a Trace? OpenTelemetry (Observability Primer, 2022) itself says the following:

"A Distributed Trace, more commonly known as a Trace, records the paths taken by requests (made by an application or end-user) as they propagate through multi-service architectures, like microservice and serverless applications. Without tracing, it is challenging to pinpoint the cause of performance problems in a distributed system."

So a Trace is like the thread or route a request or operation takes through services. Such a Trace in turn consists of Spans. A Span can be seen as the building blocks of a Trace and shows, for example, how long a function takes to complete.

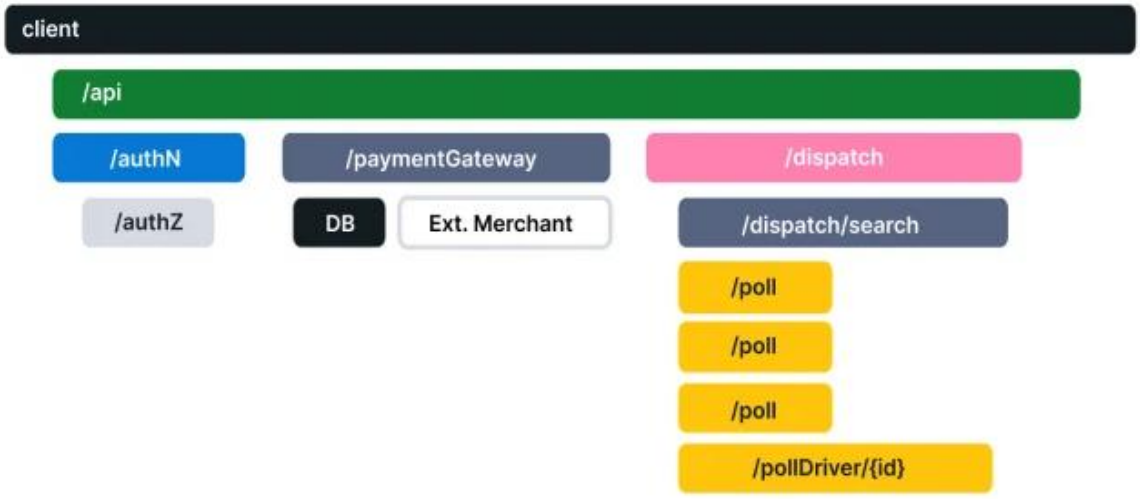

Figure 1 A Trace (client) with Spans shown below in the form of a waterfall-diagram

Observability back-end.

Now despite OpenTelemetry being, and providing, a lot of things it does not provide you with a back-end in which you can see the output of your traces. However for back-end solutions there are more options than for tracing itself. I chose the following two popular tracing back-ends and with the help of online articles (Berman, 2018) compared them:

Zipkin:

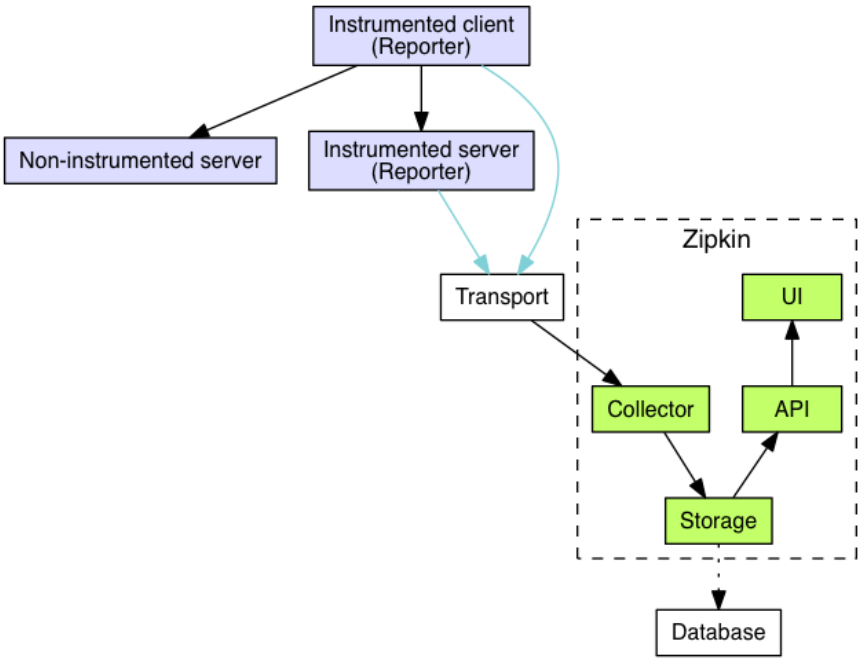

Figure 2 Zipkin Architecture

Zipkin is written in Java and uses a centralised collector model as shown in the diagram above. This makes deployment easier for the entire system but the system itself is less flexible and scalable. Zipkin supports most languages and has official support for popular libraries with the less popular libraries being supported by community development.

Jaeger:

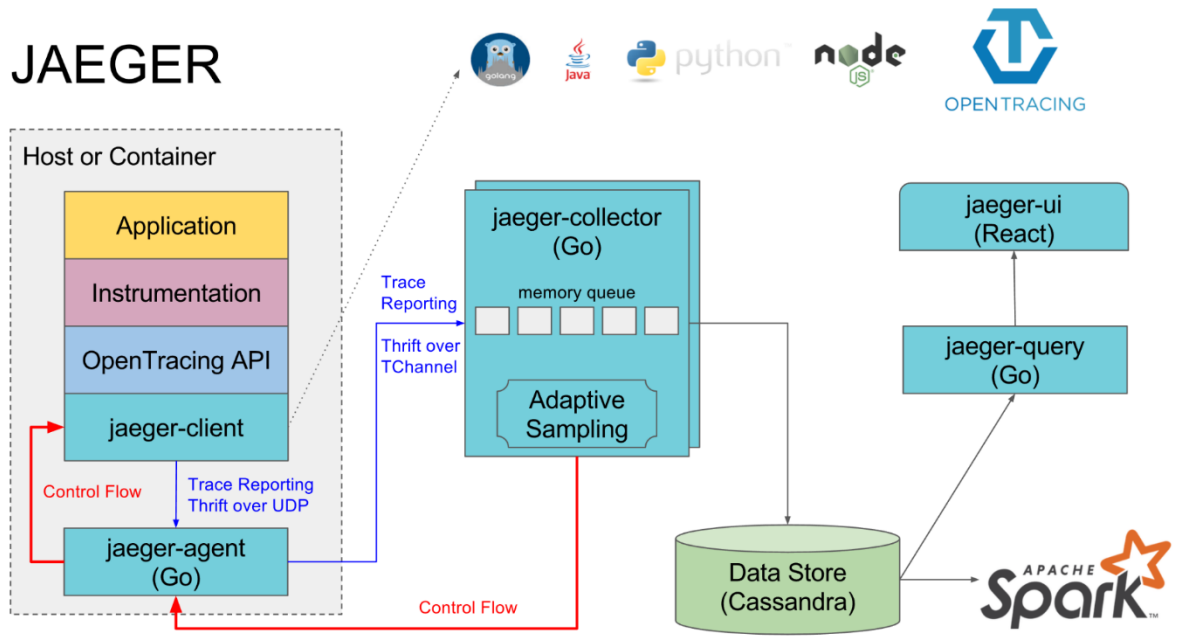

Figure 3 Jaeger Architecture

Jaeger is written in Go and uses a decentralised collector model as shown in the diagram above. This model makes it easier to deploy and scale Jaeger via Kubernetes. Jaeger, like Zipkin, supports most languages but Jaeger supports OpenTracing in its entirety in terms of libraries so that just a bit more libraries are supported. Jaeger is also part of the Cloud Native Computing Foundation, indicating good integration with cloud platforms.

In addition, Jaeger is very easy to set up in the datastore. With Jaeger, setting up with a popular datastore such as ElasticSearch or Cassandra is virtually no work at all. Jaeger also has out of the box support for gRPC which makes it particularly useful in applications I am making this semester.

To top it all off, Jaeger can even integrate with Zipkin! Should Jaeger ultimately not be the ideal choice, it is always possible to keep part of the Jaeger implementation and still use Zipkin.

2. How to implement a tracing solution?

As eluded to earlier implementing a tracing solution requires three things:

1. A tracing or instrumentation library/framework
2. Instrumented code
3. A back-end to store traces in.

I want to see how easy it is to implement a basic tracing solution.
Doing my research I found a few options:

A. Tracing and outputting to the STDOUT
B. Tracing and outputting to the jaeger exporter
C. Tracing and outputting using the gRPC exporter and the Jaeger all in one solution.

Tracing and outputting to STDOUT.

After implementing OpenTelemetry and instrumenting a basic piece of code, I actually got a text file from the STDOUT exporter. This does create a Trace with Spans but unfortunately the STDOUT exporter does not automatically show how long the Span was running. This can be solved but that requires modification to the instrumentation of the code which requires more work than it is worth for a STDOUT exporter. For this reason, the STDOUT exporter was dropped as an option and I switched to a new exporter.

Tracing and outputting to the jaeger exporter.

Since the STDOUT does not provide timestamps and thus does not meet the requirements, I went to see if the Jaeger Exporter itself has more functionality in the exporter that does provide timestamps. After exchanging the telemetry library from stdout to Jaeger, I had to rewrite the code for the traces and integrate it with the command line. This never quite worked out in the end. The exporter did not provide timestamps and no longer provided the full trace. The reason for this I couldn't find but the lack of timestamps in itself made me quit this solution as well.

Tracing and outputting using gRPC export and Jaeger all in one.

The suspicion I got after the two previously failed solutions was that the timestamps in a trace were not properly processed by the local environment. For this reason, I still decided to use a WebUI component. The idea now was to use an exporter that could message to a hosted collector. So this meant I needed another exporter and had to look for a WebUI, Collector and datastore component solution.

The final solution for the part that needed to be deployed was a Jaeger all-in-one container. This docker container combines the Jaeger agent, collector and UI components allowing everything to be controlled through one docker file. The choice of exporter for the Jaeger all in one solution was simple. I use gRPC in my individual project so I used the OpenTelemetry gRPC exporter.

Finally, a datastore had to be arranged. There are many options here but I chose what I believed to be the least effort to setup which was ElasticSearch.

Implementation can be found here.

3. Can tracing help detect bottlenecks.

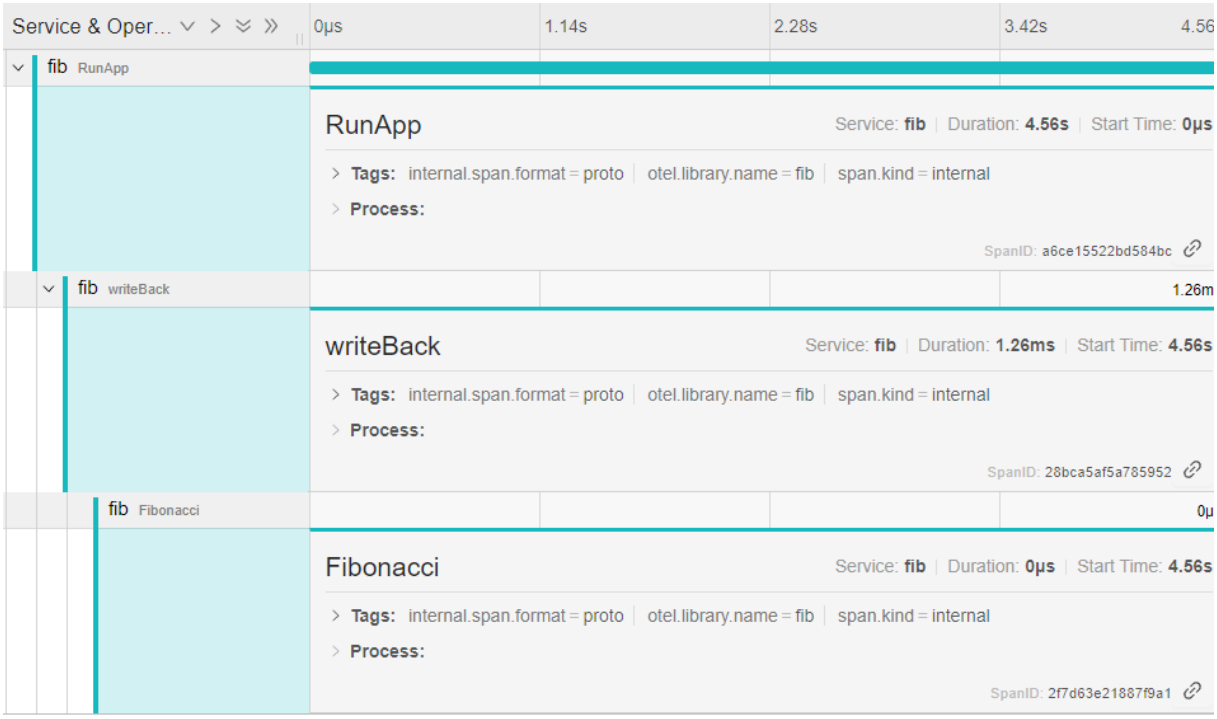

Figure 4 Jaeger UI showing the trace of the PoC

Here is the output. The Fibonacci program logs its steps and jaeger shows how long it takes. To make it a bit more extreme in can add a delay timer in one part of the program:

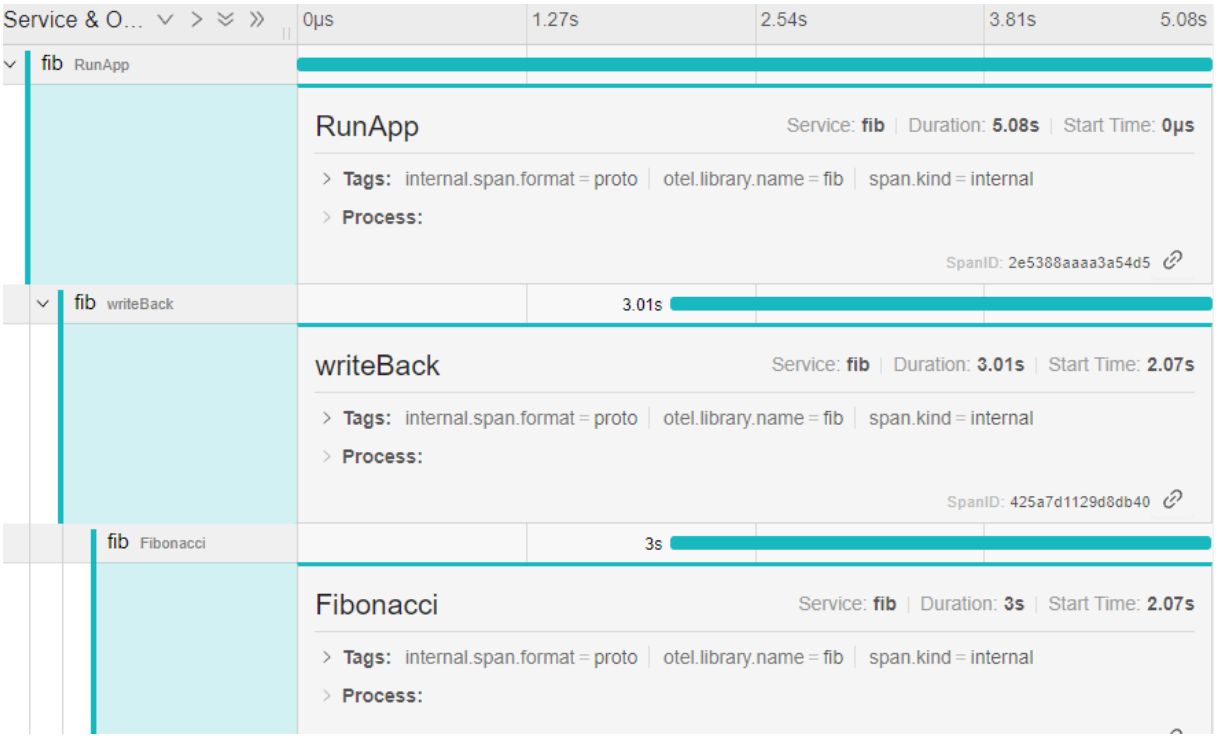

Figure 5 Jaeger UI showing the trace of the PoC with artifical delay

We can now clearly see where in the program so much time is being lost! A simple writeback shouldn't take 3+ seconds after all!

I discussed these results with my peers and asked them if they felt that this would help them discover bottlenecks. All of them agreed that this would help them figure out where bottlenecks occur.

Conclusion.

Tracing is a valuable tool that can help detect bottlenecks. However this does require a developer to setup the appropriate infrastructure and instrument their code. My prototypes showed me that setting up a quick Jaeger all-in-one solution for infrastructure is rather easy, coupling a backend database shouldn't be a huge issue but instrumenting code is a bit of work.

If a team decides that finding bottlenecks and having good visibility of their code is important enough then it is probably worth your time to instrument your code and setup tracing.

Sources

Berman, D. (2018, 12 juli). *Zipkin vs Jaeger: Getting Started With Tracing*. Logz.io. https://logz.io/blog/zipkin-vs-jaeger/

How to Implement Tracing in a Modern Distributed Application. (2018, 22 november). Cossack Labs. https://www.cossacklabs.com/blog/how-to-implement-distributed-tracing/

Jaeger architecture. (z.d.). Logz.io. https://logz.io/blog/zipkin-vs-jaeger/

Observability Primer. (2022, 1 oktober). OpenTelemetry. https://opentelemetry.io/docs/concepts/observability-primer/

What is Observability? A Beginner's Guide. (z.d.). Splunk. https://www.splunk.com/en_us/data-insider/what-is-observability.html

Zipkin architecture. (z.d.). logz.io. https://logz.io/blog/zipkin-vs-jaeger/

Waterfall tracing. (z.d.). Opentelemetry.io. https://opentelemetry.io/docs/concepts/observability-primer/#distributed-traces