Distributed Request Tracing using Zipkin and Spring Boot Sleuth

Mallanna S. D.
Department of Telecommunication,
Siddaganga Institute of Technology, Tumakuru,
Karnataka

Devika M.
Department of Electronics and Communication,
Brindavana college of engineering, Bangalore,
Karnataka

ABSTRACT

Modern cloud computing systems have brought scalability and agility by enabling the creation of microservices with the implementation of the service-oriented architecture (SOA) or a fine-grained microservice architecture (MSA). While the creation of microservices provides the agility for the development, it also brings complexity to the flow request and response tracing system-wide. Isolation of faults and the origin of the issue becomes a daunting task. Since the services can potentially be distributed on multi cloud environments, tracing the flow and sequence of calls in the inter service communication is quite challenging. Without the innovative mechanisms, the service faults cannot be located. Request tracing is a concept where a unique identifier is tagged at the origin and persisted all the way in the flow of information till the life cycle of the request ends. Request tracing is complex but can be possible with the help of some distribution tracing software. This paper attempts to throw some light on the flow of data in a distributed service environment and innovative approaches for tracing the request flows.

General Terms

Distributed Systems, Cloud Computing, Micro-Services.

Keywords

Request Tracing, Microservices, Service Decomposition, Spring Boot, Java, Sleuth, Brave and Zipkin.

1. INTRODUCTION

With the introduction of the modern cloud computing systems, creating, standing up, and deploying the services became a 'walk on the cake' easy process. It in-turn increased the velocity of building the services and delivering the newly developed features in a smaller batch with less or no effect on the rest of the deployable microservices [6] [7].

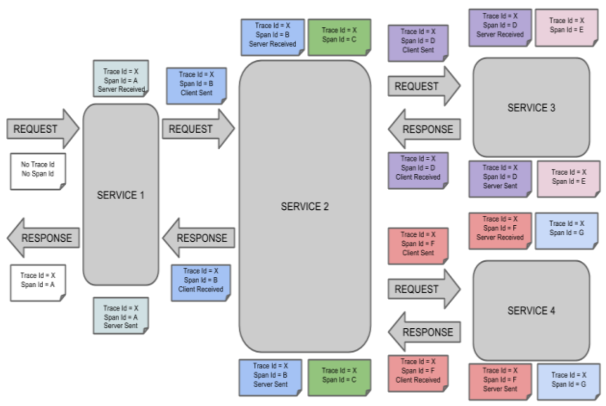
While creating the distributed microservices brought the agility to the development of the business applications it also brought complexity to infrastructure and the maintenance as explained by authors like Chaitanya K. Rudrabhatla [5]. When all the application functionality is built into single monolithic unbroken deployable .war or .ear archives, it is much easier to debug the applications.

In the case of a monolithic application intercepting and tracing any problem much easy. Is a request getting dropped? It's in the monolithic application. Isn't a component not returning an expected result? It's in the monolith. Is there a memory leak issue? It's in the monolith. With the introduction of the broken distributed microservices everything changes.

In monolith application, all the requests are handled on the single part of the application however once monolithic applications are broken down into multiple microservices either based on the Single Responsibility Principle (SRP) or the Common Closure Principle (CCP), the request can enter service and traverse across multiple services before returning the response to the API user. For example, when using a retails system like amazon, a user needs to add the production into a cart and then need to check out from the cart and finally make the payment. When a user is trying to pay for the product, a payment request can enter from a service endpoint, let's say from the service (S1) and can make multiple internal calls with the other services to check various details like availability of the product, deliverability of the product, etc., let's say between micro-services(S2-Sn) and finally returning the response to the S1. Adding to it, the cloud-first application supports dynamic scalability and load balancing. The dynamic scalability with load balancing needs to have multiple instances of the same services running to service the incoming request. In such distributed systems, debugging for bug becomes extremely difficult as the request flow must be traced across multiple microservices and multiple instances of the same services [8]. Distributed tracing helps pinpoint the problem where failure occurs. Systems behave differently under load and scale. Setting a context helps to identify the nature of the requests as they pass through a service. As the Josh Long and Kenny Bastani mentioned in the book Cloud Native Java [1] - You can't trace bugs in a system until you've established a baseline for what normal is.

In theory, request tracing is easy. Attach a unique identifier at the beginning of the request and keep passing it wherever the request flows.

Below image from the documentation [2] better illustrate the complexity that a tracing involves on a distributed system.



1.1 Who can use the Distributed Tracing?

Distributed tracing or distributed request tracing is such a useful tool for the IT and DevOps teams. It helps them to monitor and debug the failures more efficiently.

2. TRACER

2.1 Terminology

a) Trace ID: Unique identifier that represent each incoming request.

b) Span: As we discussed above, a request can flow across multiple service, say, 1 to N. N being any number. A Span of a request shows the its journey on number of distributed services its covered and the response it returned finally.

c) Tags: These are the used definer annotations of a span to query and filter on it.

d) Logs: Documentation of specific movement or the event.

e) Sampling: Number for Tracer to keep.

2.2 Tracers

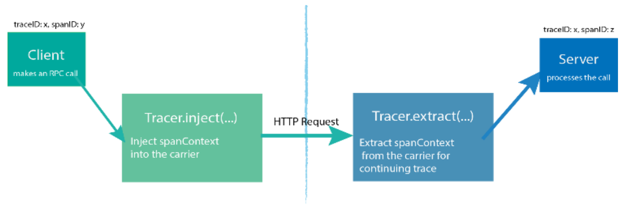
Tracer created and lives within the application container to record the requests metadata and timing when it happens. A tracer's main job is

- Capture the context and pass metadata in it to the next service.
- Encoding and decoding the context metadata over the network calls.
- Causality tracking: Track the Parent-Child, forks, Joins.

At the beginning of the request, a Tracer Interface should create a span and it should it should Extract the metadata (Incoming) or Inject the metadata (outgoing).

Tracer should also record the events information such as the component name, API endpoints, etc.

An illustration shows on OpenTracing[3]



3. ZIPKIN AND SLEUTH

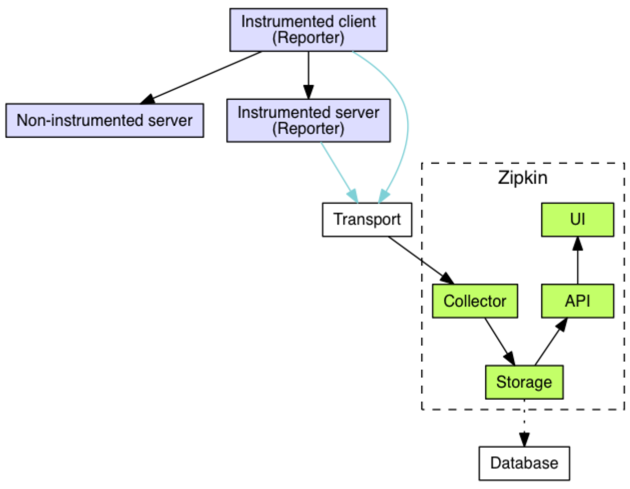
Zipkin is one of the OpenTracing supported distributed Tracer System. It helps gathering the metadata needed to trace and troubleshoot the latency problems on a distributed service architecture.

3.1 Zipkin Architecture Overview

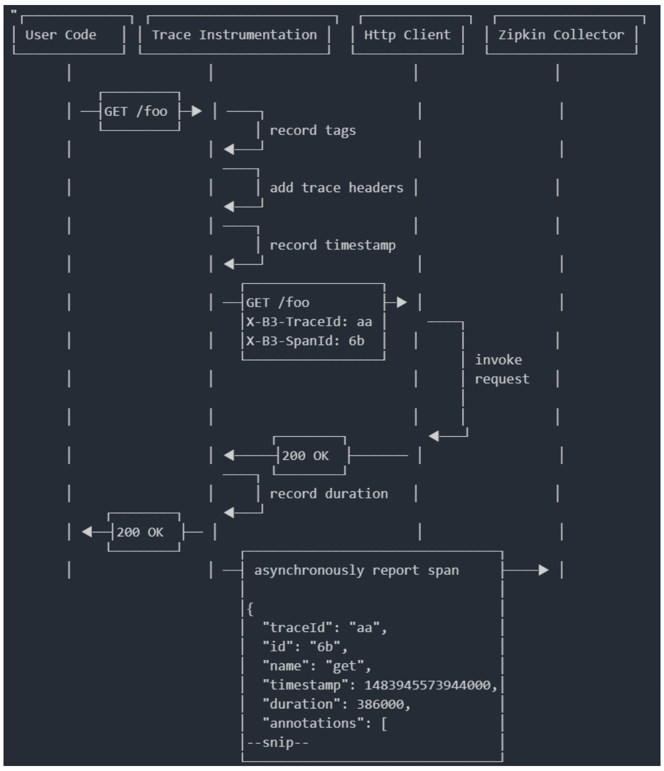
As mentioned on 2.2, tracer live insider the container capturing the metadata. Zipkin uses the safe instruments to attach the propagationID to tell the receiver that there is a trace in progress. Once completed the flow, spans are reported to Zipkin for logging and analysis.

Example[4]: When an operation is traced, when an outgoing http request is formed, a few headers are added to propagate ID.

Component which sends the span and metadata to the Zipkin Collector is called as Reporter. Zipkin Collectors write the trace data to storage which later can be queried for the analysis. An illustrative architecture diagram[4] is shown below.



Here is an example to show the complete data flow –



```
┌───────────┐ ┌─────────────────────┐ ┌─────────────┐ ┌──────────────────┐
│ User Code │ │ Trace Instrumentation │ │ Http Client │ │ Zipkin Collector │
└───────────┘ └─────────────────────┘ └─────────────┘ └──────────────────┘
      │                 │                        │                  │
      │ ┌─────────┐     │                        │                  │
      │ ──┤GET /foo├──▶ │ ────┐                  │                  │
      │ └─────────┘     │     │ record tags      │                  │
      │                 │ ◀───┘                  │                  │
      │                 │ ────┐                  │                  │
      │                 │     │ add trace headers│                  │
      │                 │ ◀───┘                  │                  │
      │                 │ ────┐                  │                  │
      │                 │     │ record timestamp │                  │
      │                 │ ◀───┘                  │                  │
      │                 │ ┌─────────────────┐    │                  │
      │                 │ │GET /foo         ├──▶ │                  │
      │                 │ │X-B3-TraceId: aa │    │ ────┐            │
      │                 │ │X-B3-SpanId: 6b  │    │     │            │
      │                 │ └─────────────────┘    │     │ invoke     │
      │                 │                        │     │ request    │
      │                 │                        │     │            │
      │                 │                        │ ◀───┘            │
      │                 │        ┌────────┐      │                  │
      │                 │ ◀──────┤200 OK  ├───── │                  │
      │                 │        └────────┘      │                  │
      │                 │ ────┐                  │                  │
      │                 │     │ record duration  │                  │
      │   ┌────────┐    │ ◀───┘                  │                  │
      │ ◀─┤200 OK  ├─── │                        │                  │
      │   └────────┘    │                        │                  │
      │                 │ ┌────────────────────────────────┐        │
      │                 │ ─┤ asynchronously report span    ├──────▶ │
      │                 │ │                                │        │
      │                 │ │{                               │
      │                 │ │  "traceId": "aa",              │
      │                 │ │  "id": "6b",                   │
      │                 │ │  "name": "get",                │
      │                 │ │  "timestamp": 1483945573944000,│
      │                 │ │  "duration": 386000,           │
      │                 │ │  "annotations": [              │
      │                 │ │--snip--                        │
                          └────────────────────────────────┘
```

3.2 Components of Zipkin

There are 4 main components

- Collector: Is responsible for validating, storing, and indexing the data for the lookups.
- Storage: Is where the indexed data is stored. Supported DBs - Cassandra, ElasticSearch and MySQL
- Query Service: Tool to query and read the indexed data.
- Web UI: A nice interface for viewing the trace data.

3.3 Brave

Brave is a distributed tracing instrumentation library which send the trace data to Zipkin. It basically intercepts the production incoming request to gather timing data, correlate and propagate the trace contexts.

The data collected would be typically sent to Zipkin server or can be sent over to AWS Cloudwatch/X-ray using the customized libraries.

Brave provides a dependency free Java library which includes the filters for Servlet and correlate for Log4J.

3.4 Spring Boot Sleuth

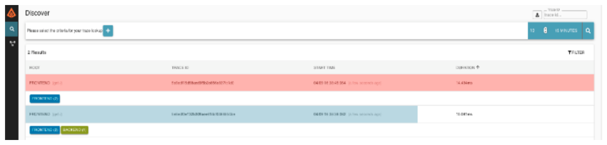
When build the cloud native Java application, Spring Boot provides the best solutions and standup and Deploy the micro service with the help of its auto-configurations. Spring Boot Sleuth is no exception to it. Spring boot Sleuth has the auto-configurations for Distribution Tracing. Sleuth is built on top the Brave the tracer library.

Spring Boot Sleuth autoconfigures everything that IT or DevOps needs to get started. Auto Configurations include

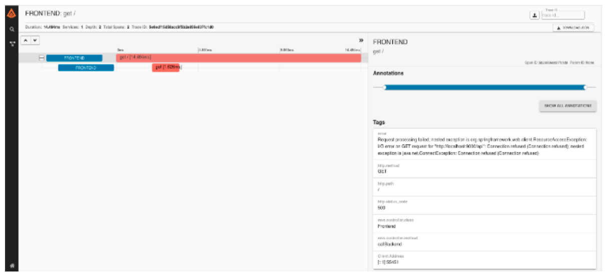
- Span: where the trace data is sent to.
- Sampling: number of tracers to setup.
- Baggage: Remote Fields
- Libraries Traced.

Sleuth sets up the instrumentation not just to track timing but to track the errors as well so that they can be analyzed and correlated with Log4J.

Here is the example of Web UI showing the trace.



Request is red color indicate the errored or failed. To see the further details, one can click on the error trace.



error
Request processing failed; nested exception is org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://localhost:9000/api": Connection refused (Connection refused); nested exception is java.net.ConnectException: Connection refused (Connection refused)

http.method
GET

http.path
/

http.status_code
500

mvc.controller.class
Frontend

mvc.controller.method
callBackend

Client Address
[::1]:55451

4. ADDING SLEUTH AND ZIPKIN TO THE SPRING BOOT PROJECTS.

Staring up with the Sleuth and Zipkin is very easy since the Spring boot takes care of many auto configurations.

Spring Boot provides the flexibility of choosing both the all the components of the Zipkin or just the log correlation with the Sleuth.

4.1 Adding both Zipkin and Sleuth

Leaving many of the configurations to the Spring Boot, one can just add the dependency via the Maven or the Gradle to start over.For its readability and simplicity, we are showing how to add the dependency through Gradle. By adding the below code to a build.gradle file, one can start with the configuration.

```
compile "org.springframework.cloud:spring-cloud-starter-zipkin"
```

4.2 Adding Sleuth (Only Log Correlation) to the project

One can add the dependency to the project to get started with the Sleuth as below -

```
compile "org.springframework.cloud:spring-cloud-starter-sleuth"
```

Here is a sample log documented –

```
2020-07-14 14:12:45.404  INFO [zipkin-service-id,12v44d0t19e41d00,12v44d0t19e41d00,false]
```

Here Zipkin-service-id is the unique service name.

12v44d0t19e41d00 is trace ID and Span ID.

5. ACKNOWLEDGMENTS

We thank all the scholars and researchers on Reference section who provided the enough information for us to understand the Distributed Tracing systems.

6. REFERENCES

[1] Cloud Native Java: Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry. Long, J. and Bastani, K. O'Reilly Media, Incorporated. 2017. ISBN/ 9781449374648. LCCN/ 2017277168.

[2] Spring Cloud Sleuth. Adrian Cole, Spencer Gibb, Marcin Grzejszczak, Dave Syer, Jay Bryant at https://cloud.spring.io/spring-cloud-sleuth/reference/html/

[3] OpenTracing: https://opentracing.io/

[4] OpenZipkin: https://zipkin.io/

[5] Chaitanya K Rudrabhatla. A Systematic Study of Micro Service Architecture Evolution and their Deployment Patterns. International Journal of Computer Applications 182(29):18-24, November 2018

[6] S. Newman, Building Microservices. " O'Reilly Media, Inc.", 2015.

[7] D. Namiot and M. Sneps-Sneppe, " On micro-services architecture," International Journal of Open Information Technologies, vol. 2, no. 9, 2014.

[8] Xiaolu Zhou, Yang Xie, Hui Xie, Guihua Wang, "Research on Session Sharing of Distributed Application Service Technology", Advanced Information Technology Electronic and Automation Control Conference (IAEAC) 2018 IEEE 3rd, pp. 894-899, 2018.