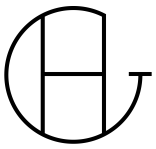


HoGent

Faculteit Bedrijf en Organisatie

Tracing in microservices met Spring Cloud Sleuth en Zipkin

Frederic Everaert

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Harm De Weirdt
Co-promotor:
Geert Vandensteen

Instelling: —

Academiejaar: 2016-2017

Eerste examenperiode

Faculteit Bedrijf en Organisatie

Tracing in microservices met Spring Cloud Sleuth en Zipkin

Frederic Everaert

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Harm De Weirdt
Co-promotor:
Geert Vandensteen

Instelling: —

Academiejaar: 2016-2017

Eerste examenperiode

Samenvatting

In een microservice architectuur is het vaak moeilijk om het overzicht te bewaren op het volledige systeem. Een oplossing hiervoor is tracing. In dit onderzoek wordt specifiek gekeken naar Spring Boot microservices en hoe performant het is om tracing toe te voegen met behulp van Sleuth en Zipkin. Het debuggen van microservices ligt vaak ook niet voor de hand. Om dit aan te pakken kan gebruik gemaakt worden van enkele logging tools, waar in dit onderzoek de performantie van twee log verzamelaars met elkaar vergeleken worden.

In dit onderzoek worden de volgende twee vragen beantwoord:

- Wat zijn de performantiegevolgen die tracing met zich meedraagt in een Spring Boot microservices omgeving? Hoe kan sampling hierbij helpen?
- Om logs van een Spring Boot microservice naar ElasticSearch te sturen worden Logstash en Fluentd bekeken. Is er een performantieverschil?

Het onderzoek werd uitgevoerd op 3 verschillende opstellingen: een lage complexiteit setup, een gemiddelde en een hoge. Voor de details zie figuren 3.1, 3.2 en 3.3. Elke setup werd getest met laag tot hoog netwerkverkeer.

De verwachting was dat tracing het systeem wel degelijk extra zou belasten, maar het was voor dit onderzoek nog onduidelijk hoe dit zou zijn in een Spring Boot omgeving met Sleuth en Zipkin. Het resultaat van dit onderzoek was dat het toevoegen van tracing het systeem wel degelijk extra belast, maar dit enkel op te merken zal zijn in een hoge complexiteit setup met hoog netwerkverkeer. In dat geval werd genoteerd dat het instelling van een sampling frequentie lager als 50% de performantie sterk verbetert.

De verwachtingen voor performantie van Logstash tegenover Fluentd in een Spring Boot

omgeving waren voor dit onderzoek compleet onduidelijk. Het resultaat van dit onderzoek was dat in een Spring Boot omgeving Fluentd iets performanter is dan Logstash. Het verzamelen van logs met Logstash of Fluentd heeft niet zo'n grote impact op de performantie van het volledige systeem.

Voorwoord

Deze bachelorproef dient als eindwerk voor de opleiding toegepaste informatica aan Hogeschool Gent.

Vorige zomervakantie deed ik een vakantiejob bij Synthetron, waar ik werkte met Spring Boot en microservices. Dit was een heel leerrijke ervaring en ik besloot een onderwerp te zoeken in de wereld van microservices. Het was mijn co-promotor Geert Vandensteen, tevens mijn opdrachtgever van bij Synthetron, die mij voorstelde om tracing te gaan onderzoeken. De uiteindelijke focus kwam dan te liggen op de performantiegevolgen die daarmee gepaard gingen.

Ik wil zowel mijn promotor Harm De Weirdt, als mijn co-promotor Geert Vandensteen bedanken voor de begeleiding en feedback die ik kreeg bij het voltooien van deze bachelorproef.

Inhoudsopgave

1. Inleiding

Monolithische architecturen ruimen steeds meer baan voor microservice architecturen of kortweg microservices. Een monolitische applicatie is gebouwd als een enkelvoudige, zelfstandige eenheid. Bij een client-server model, kan bijvoorbeeld de applicatie langs de server zijde bestaan uit een enkele applicatie die de HTTP requests behandelt, logica uitvoert en data ophaalt of bijwerkt in de database. Het grote probleem van monolitische applicaties is dat ze moeilijk te onderhouden zijn. Een kleine verandering in een bepaalde functionaliteit van de applicatie kan ervoor zorgen dat andere delen van de applicatie ook bijgewerkt moeten worden. De volledige applicatie moet opnieuw gebuild en gedeployed worden. Als een bepaalde functionaliteit gescaled moet worden, moet de volledige applicatie gescaled worden. Microservices bieden hiervoor een oplossing, aangezien de verschillende functionaliteiten elk een op zich zelf staande service kunnen vormen.

Bij microservices schieten echter de traditionele tools voor debugging tekort. Een enkele service kan niet het volledige beeld geven over bijvoorbeeld de performantie van de applicatie in zijn geheel. Om dit te verhelpen, kunnen requests getraced worden. Een trace is de volledige reisweg van een request die spans bevat voor alle doorkruiste microservices. Een span bestaat uit tags of metadata zoals de start- en stoptijdstippen. Deze data kan dan verzameld worden om een volledig beeld van het gedrag van de applicatie te geven.

De bedoeling van dit onderzoek is om de meerwaarde van tracing in microservices aan te tonen en ook om eventuele tekortkomingen vast te stellen. Er wordt specifiek gekeken naar tracing van requests met behulp van Spring Cloud Sleuth en Zipkin. In de praktijk gaat men niet alle requests gaan tracen om onnodige overhead te vermijden. Beslissen welke requests getraced worden en welke niet wordt sampling genoemd. Wat kunnen bijvoorbeeld enkele interessante use cases zijn waarbij er op een intelligente manier gesampled kan worden? Ten slotte visualiseert Zipkin enkel de verschillende traces, maar niet de log berichten,

deze worden dan ook nog niet verzameld. Om logs te verzamelen en visualiseren wordt gekeken naar ElasticSearch, Logstash / Fluentd en Kibana. Er wordt aangetoond hoe deze tools in combinatie met Zipkin helpen om de complexiteit van microservices in bedwang te houden.

1.1 Stand van zaken

Onderzoeken rond distributed tracing systemen zijn schaars, maar de technologieën die gebruikt worden in dit onderzoek, namelijk Spring Cloud Sleuth en Zipkin, zijn gebaseerd op Google Dapper. (Sigelman e.a., 2010) Deze technische paper beschrijft in detail de werking van Google's distributed tracing systeem. Net zoals Dapper gebruikt Sleuth een annotatie gebaseerde methode om tracing toe te voegen aan requests. De terminologie is volledig overgenomen. Een trace bevat meerdere spans voor elke hop naar een andere service. Spans die dezelfde trace id bevatten, maken deel uit van dezelfde trace. Een groot ontwerpdoel van Dapper was om de overhead voor tracing zo laag mogelijk te houden. Sampling werd hierdoor geïntroduceerd. In een eerste productie gebruikten ze eenzelfde sampling percentage voor alle processen bij Google, gemiddeld één trace per 1024 kandidaten. Dit schema bleek heel effectief te zijn voor diensten met veel verkeer, maar bij diensten met minder verkeer gingen er zo interessante events verloren. Als oplossing werd toegelaten dat dit sampling percentage wel aan te passen was, ook al wou Google met Dapper dit soort manuele interventie vermijden. Spring Cloud Sleuth neemt dezelfde instelling over en laat ook toe om zelf sampling in te stellen.

1.2 Probleemstelling en Onderzoeksvragen

Het is voorlopig nog onduidelijk welke performantiegevolgen tracing met zich meedraagt in een Spring Boot microservices omgeving. Deze worden in dit onderzoek bekeken door de sampling van Sleuth in te stellen.

Om logs van een Spring Boot microservice naar ElasticSearch te sturen worden Logstash en Fluentd bekeken. Is er een performantieverschil?

1.3 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 worden de gebruikte technologieën in dit onderzoek besproken. In Hoofdstuk 3 wordt de methodologie toegelicht en worden antwoorden gezocht op de onderzoeksvragen. In Hoofdstuk 4, wordt tenslotte de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen.

2. Gebruikte technologieën

2.1 Spring en Spring Boot

Dit onderzoek maakt gebruik van Spring Boot om microservices op te stellen. Hieronder volgt een inleiding tot het Spring framework en het verschil tussen Spring en Spring Boot.

Het Spring framework is een open source framework gericht op applicatieontwikkeling in Java (Pivotal Software, 2014). Het bekendste onderdeel van het framework is het Inversion of Control (IoC) principe, ook gekend als dependency injection. Dit betekent onder andere dat een object geannoteerd kan worden met `@Component` en Spring zal het object aanmaken, alle nodige velden vullen en het object toevoegen aan de context van de applicatie. Op deze manier is het mogelijk om verschillende objecten toe te voegen aan de context, zodat ze onderling gemakkelijk samenwerken. Het IoC principe vergemakklijkt hierdoor de manier om andere bibliotheken te integreren in de applicatie.

Spring Boot kan gezien worden als een uitbreiding op het Spring framework. Het maakt het gemakkelijk om op zichzelf staande, productiewaardige Spring applicaties te maken waar het niet nodig is om veel aan de configuratie te sleutelen (Pivotal Software, 2015a). Het mantra van Spring Boot is dan ook *convention-over-configuration*. Dit is bijzonder handig voor microservices omdat bijvoorbeeld een eenvoudige REST applicatie aangemaakt kan worden in minder dan 20 lijnen code.

Listing 2.1: eenvoudige Spring Boot REST app

```
@Controller
@EnableAutoConfiguration
public class SampleController {
```

```
    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SampleController.class, args);
    }
}
```

De `@Controller` annotatie zorgt ervoor dat de klasse gebruikt kan worden door Spring MVC om web requests te behandelen. `@RestController` kan vanaf Spring 4.0 gebruikt worden, deze combineert de annotaties `@Controller` en `@ResponseBody`. De laatste annotatie is handig om `return` waarden van requests op te vangen in een zelfgedefinieerde Java klasse. `@EnableAutoConfiguration` zegt dat Spring Boot de applicatie automatisch moet configureren op basis van de toegevoegde bibliotheken. Ten slotte is er nog de `@RequestMapping` annotatie die Spring vertelt dat het pad `/` gemapped moet worden op de `home` methode.

Omdat met Spring Boot zo weinig mogelijk aan de configuratie gesleuteld dient te worden, werd dit framework gekozen als platform voor de microservices in dit onderzoek.

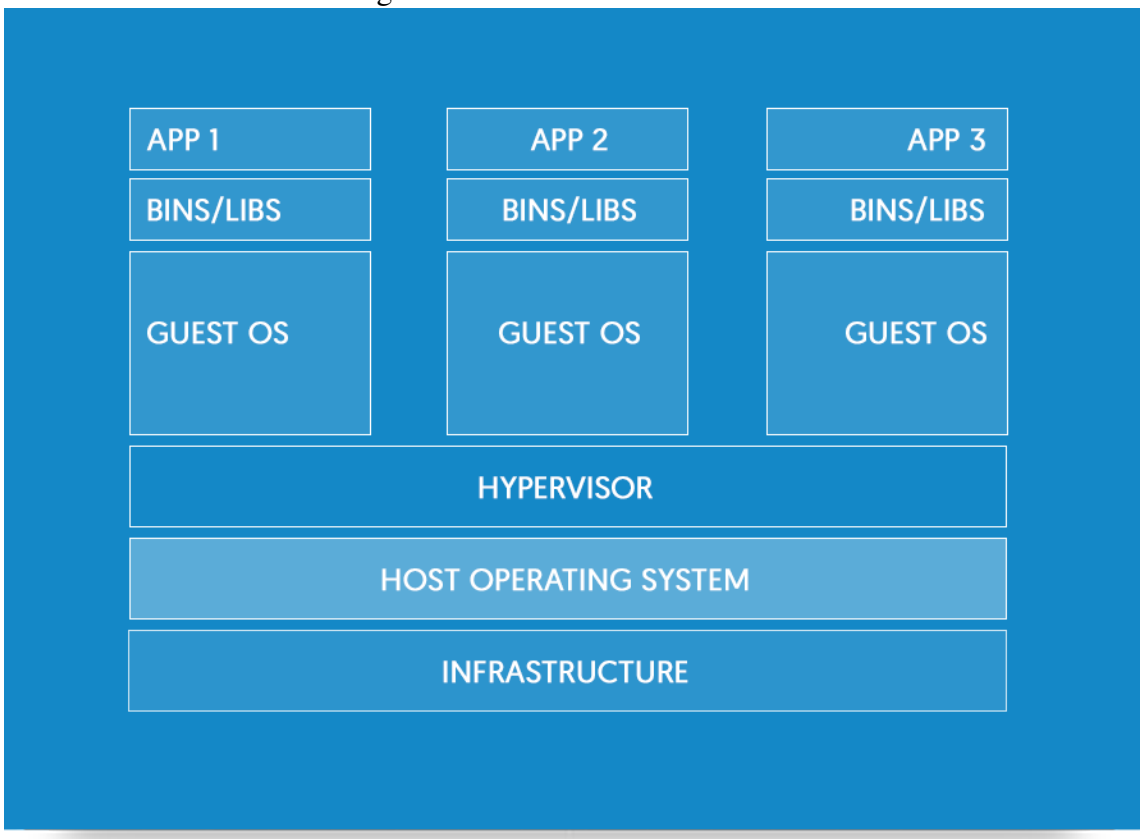
2.2 Containerisatie met Docker

Docker is een open-source platform met als doel het opzetten van applicaties te automatiseren (Docker, 2016). Elke microservice, bijvoorbeeld een Spring Boot applicatie, kan door Docker uitgevoerd worden in zijn eigen container. De verschillende containers zijn geïsoleerd van elkaar en delen enkel de minimale kernel van het besturingssysteem. Containers lijken op het eerste zicht op virtuele machines, in het opzicht dat beide geïsoleerde omgevingen zijn die beheerd worden door een controlerend proces: een container manager en hypervisor respectievelijk. Het grootste verschil tussen de twee is echter dat, voor elke virtuele machine, een volledige stack van componenten uitgevoerd dienen te worden: het besturingssysteem tot en met de applicatielaag en de virtuele hardware met netwerkkaarten, CPU's en geheugen (zie figuur 2.1).

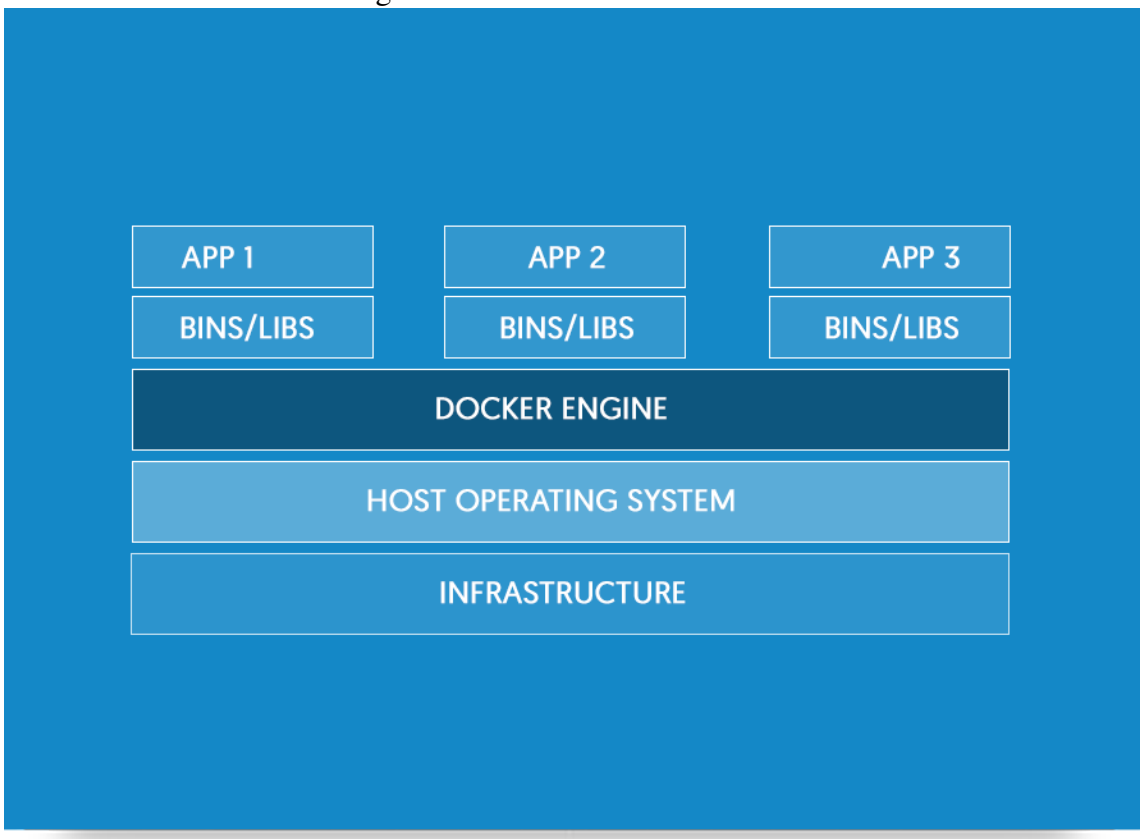
Containers daarentegen functioneren eerder als volledig geïsoleerde *sandboxes*. De containers delen met elkaar de kernel van het besturingssysteem met de aanwezige systeembronnen (zie figuur 2.2). Dit betekent dat containers veel minder zwaar zijn op het onderliggende systeem, zodat meer containers uitgevoerd kunnen worden dan virtuele machines. Een belangrijke limitatie van containers is echter dat ze enkel uitgevoerd kunnen worden in Linux-gebaseerde besturingssystemen. Dit omdat Docker gebruik maakt van kernel isolatie, wat een specifieke Linux technologie is.

Docker kan niet rechtstreeks uitgevoerd worden op Mac of Windows systemen, maar er is een eenvoudige workaround om dit te verhelpen: Docker Toolbox. Er zal eerst een Linux

Figuur 2.1: Virtuele machine stack



Figuur 2.2: Docker container stack



virtuele machine opgestart worden in VirtualBox, waarna de docker containers uitgevoerd kunnen worden in deze virtuele machine.

2.3 Tracing

Een van de uitdagingen in een microservice omgeving is om het overzicht te bewaren hoe alle verschillende services met elkaar samenwerken. Omdat elke service onafhankelijk is van de volgende, is het moeilijker om het gedrag van het volledige systeem te controleren. Zoals eerder vermeld in 1 kan tracing hierbij helpen en omdat het gaat over tracing in een microservice systeem, spreekt men ook over distributed tracing.

2.3.1 Spring Cloud Sleuth

Spring Cloud biedt een aantal bibliotheken aan zoals Config (een plaats om configuraties van verschillende microservices te beheren), Netflix (Netflix OSS integraties voor Spring Boot apps zoals Eureka) en dus ook Sleuth (Pivotal Software, 2015b). Sleuth is een distributed tracing oplossing die concepten van Dapper, Zipkin en HTrace leent.

Een trace ziet er bij Sleuth uit zoals bij Dapper. Het bevat een trace id dat gevormd wordt wanneer de eerste request gemaakt wordt. Voor elke service die de request doorkruist, wordt een span id toegekend voor die service en toegevoegd aan de trace. Zie figuur 2.3 als voorbeeld. Op de afbeelding hebben alle 5 spans dezelfde trace id (niet getoond op de figuur). Bij Sleuth is het zo dat de eerste span van een trace dezelfde id kent voor zowel span en trace id. Sleuth zal er dan voor zorgen dat die id's, samen met responstijd data, toegevoegd worden aan de headers in de request responses.

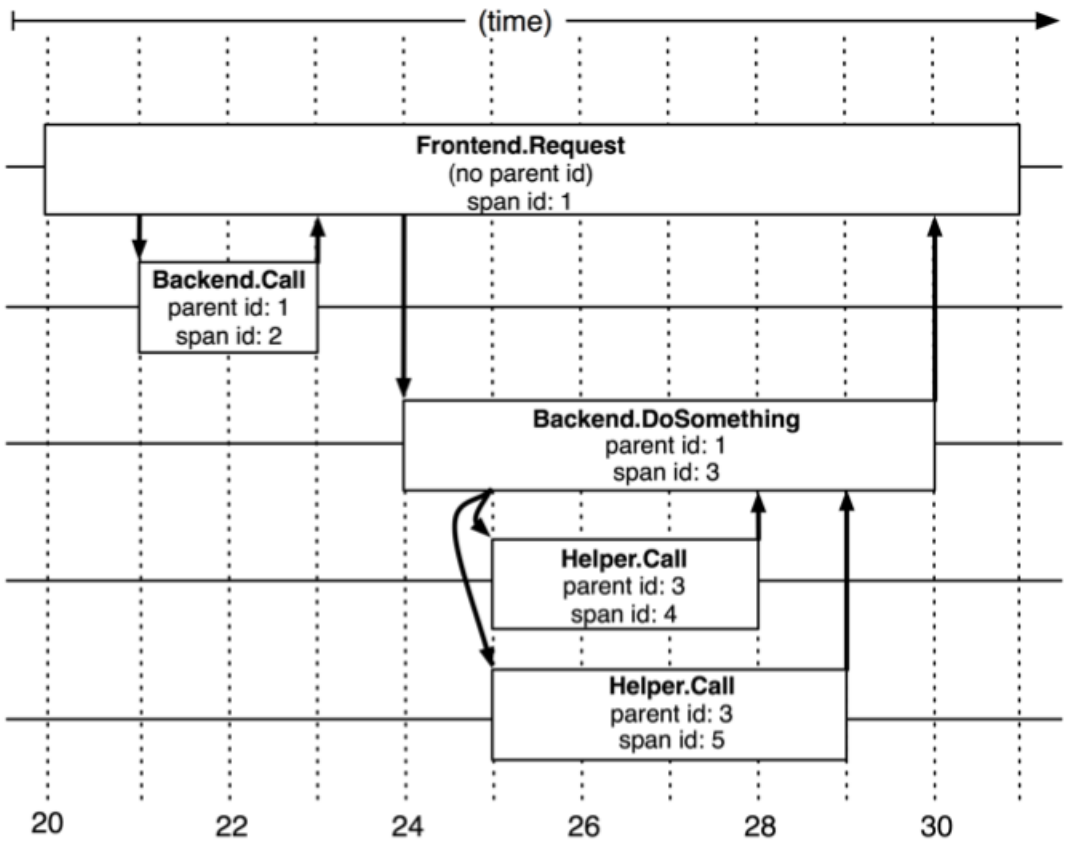
Sleuth voegt de trace en span id's ook toe aan de logging voor de microservice. Dit ziet er zo uit: `[my-service-id,73b62c0f90d11e06,73b62c0f90d11e06,false]`. `my-service-id` is de naam van de service, de eerste id duidt de trace aan, de volgende is de span en de laatste waarde geeft aan of de span geëxporteerd dient te worden naar Zipkin.

2.3.2 Zipkin

Zipkin is een open source project dat helpt bij het verzamelen en visualiseren van tracing data om responstijd problemen in microservice architecturen te kunnen detecteren („Open-Zipkin · A distributed tracing system", g.d.).

Sleuth heeft de mogelijkheid om tracing informatie te sturen naar een Zipkin server door de dependency `spring-cloud-sleuth-zipkin` toe te voegen aan de applicatie. Sleuth gaat er standaard vanuit dat de Zipkin server loopt op `http://localhost:9411`. De locatie kan echter aangepast worden door `spring.zipkin.baseUrl` toe te voegen aan de applicatie eigenschappen (application.properties).

Figuur 2.3: Voorbeeld van een trace met 5 spans



2.4 Logging

Als er iets vreemd wordt opgemerkt in de tracing informatie die Zipkin visualiseert, kan er worden afgeleid in welke service het probleem zich situeert, maar niet wat dat probleem altijd is. Om problemen op te lossen, dient de beheerder van de microservices zich te verdiepen in log bestanden. Als hulpmiddel hiervoor kan de ELK stack van pas komen. ELK staat voor ElasticSearch, Logstash en Kibana. Logstash verzamelt de logs van een microservice en stuurt deze naar een ElasticSearch server die de logs indexeert. De Kibana server is de UI die de logs uit ElasticSearch leest en ze op een gebruiksvriendelijke manier weergeeft. Met Kibana is het dan mogelijk om bijvoorbeeld alle logs van een bepaalde trace weer te geven door te filteren op trace id. Als log shipper kan ook Fluentd gebruikt worden in plaats van Logstash. In dit onderzoek wordt bekeken welke log shipper het meest performant is

3. Praktische uitwerking

3.1 Methodologie

3.1.1 Opzetten microservices

De opstelling kent de volgende microservices:

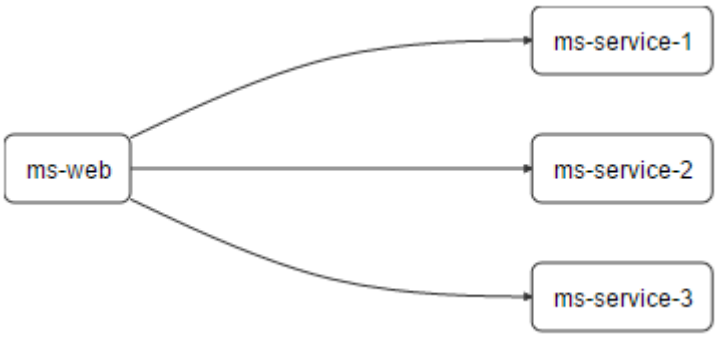
- eureka: service discovery, opdat de microservices intern met elkaar kunnen communiceren
- zipkin: Zipkin server voor tracing
- ms-web: Spring Boot app die root microservice voorstelt
- ms-service-n: Spring Boot app voor bijkomende microservices (n staat voor een cijfer van 1 t.e.m. 27)

Dit onderzoekt bekijkt drie verschillende opstellingen, geonderscheid op complexiteit, om de tracing performantie na te gaan. De setup met lage complexiteit kent in totaal 4 microservices en 2 niveau's (zie figuur 3.1). De setup met gemiddelde complexiteit kent 11 microservices en 3 niveau's (zie figuur 3.2). De setup met hoge complexiteit kent 28 microservices en 4 niveau's (zie figuur 3.3). Een request naar ms-web zal dus, afhankelijk van de setup, meerdere interne requests maken naar andere microservices. Er wordt per opstelling onderzocht wat voor performantiegevolgen tracing met zich meedraagt.

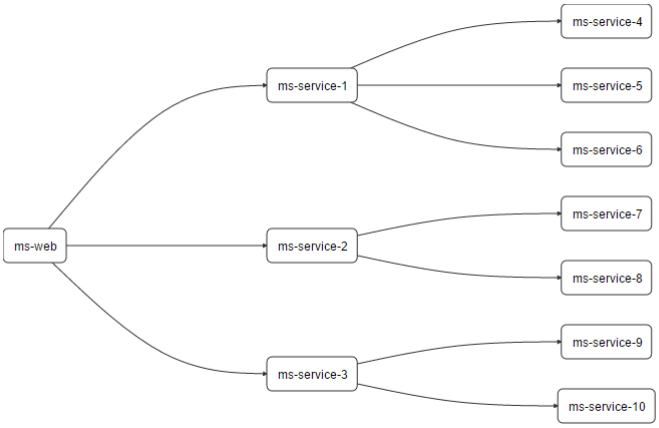
De Spring Cloud Sleuth bibliotheek wordt toegevoegd aan de ms-web en ms-service-n microservices, zodat deze services tracing info doorsturen naar de zipkin server.

Elke microservice krijgt zijn eigen Docker container, die allemaal tegelijk worden opgestart met Docker Compose (Docker Inc., 2015), dankzij het toevoegen van een docker-

Figuur 3.1: Lage complexiteit setup



Figuur 3.2: Gemiddelde complexiteit setup



compose.yml script, die alle service definities bevat.

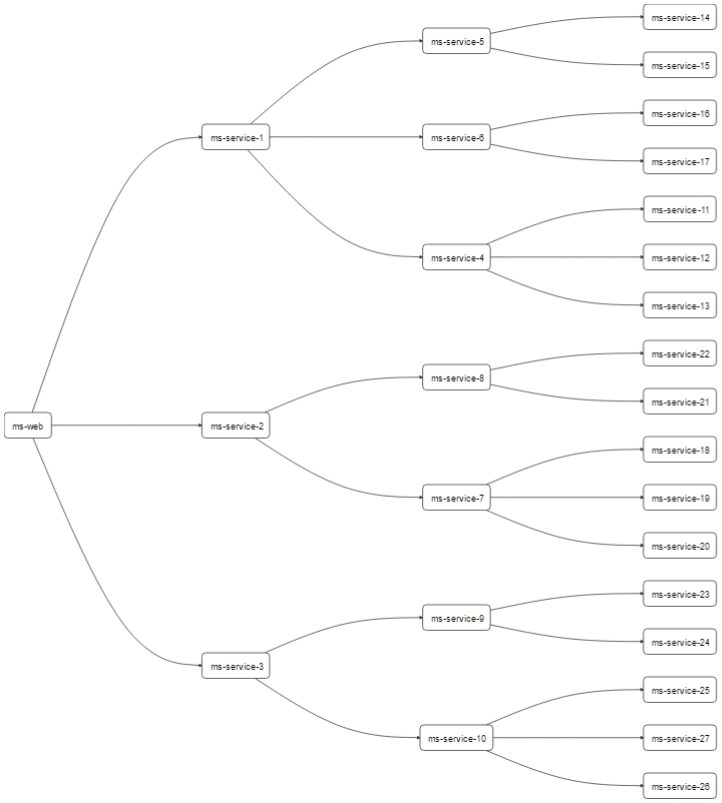
De microservices communiceren asynchroon met elkaar, zodat de ene service niet hoeft te wachten op de andere. Dit verbetert de responstijd. Vergelijk figuur 3.4 met figuur 3.5.

Om de asynchrone communicatie tussen services mogelijk te maken wordt gebruik gemaakt van `ListenableFuture` van guava (Google Core libraries voor Java) in combinatie met `TraceCallable` van Sleuth. `TraceCallable` is een wrapper van Sleuth voor een Java `Callable` die tracing informatie toevoegt. Een gewone `Callable` voert een taak uit op een andere thread en geeft een resultaat terug.

Listing 3.1: Asynchrone REST communicatie

```
@RestController
@RequestMapping("/")
public class SimpleRestController {
    private static final ListeningExecutorService executor = MoreExecutors.listeningDecorator(Executors.new

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private Tracer tracer;

    @Autowired
```

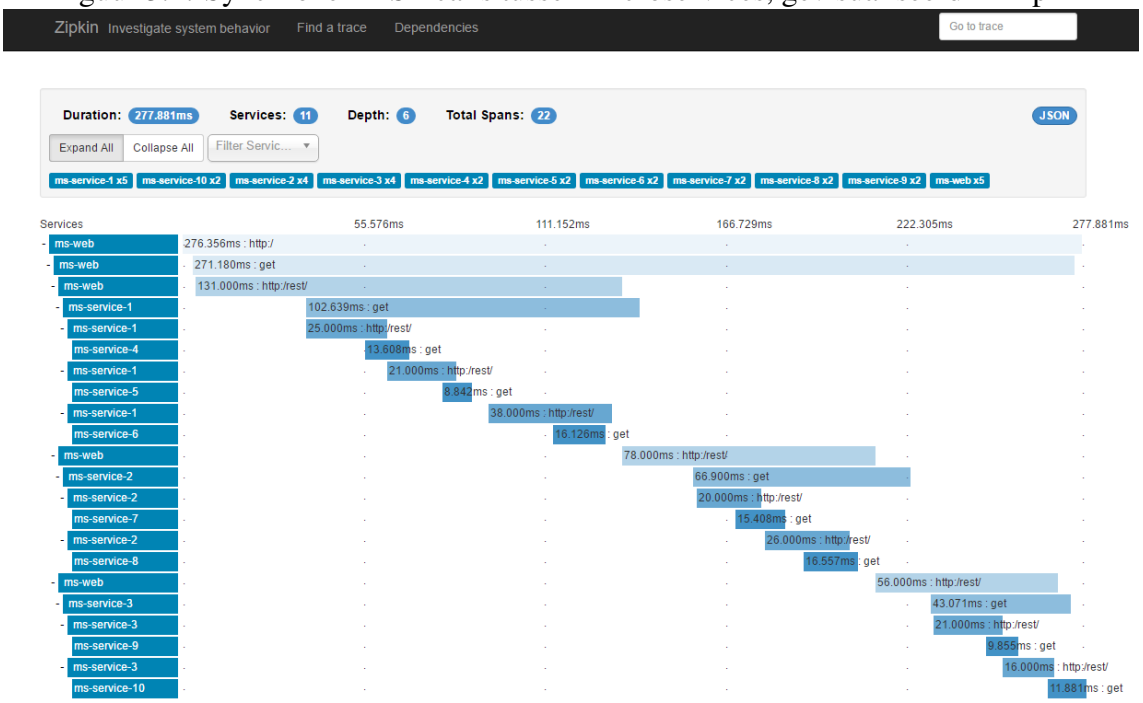
Figuur 3.3: Hoge complexiteit setup



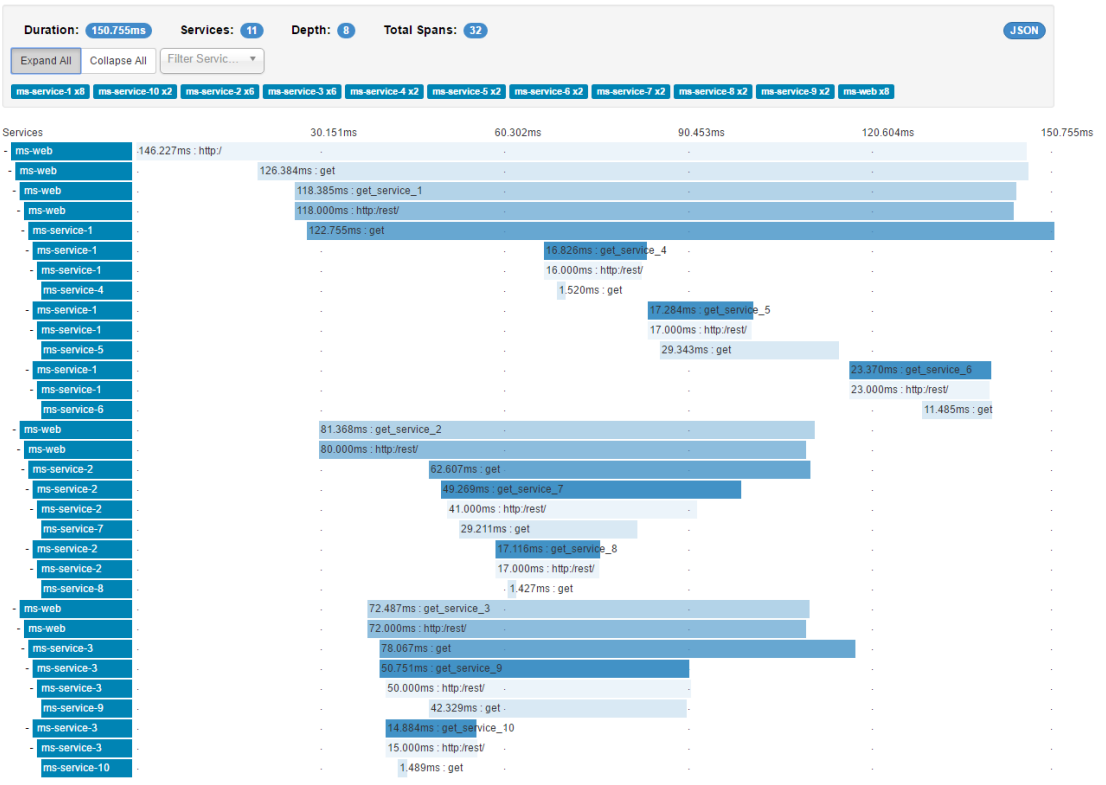

```
    private SpanNamer spanNamer;

    @RequestMapping(method = RequestMethod.GET)
    public String get() throws ExecutionException, InterruptedException {
        Callable<ResponseEntity<String>> callable1 = new Callable<ResponseEntity<String>>() {
            @Override
            public ResponseEntity<String> call() throws Exception {
                return requestService("http://ms-service-1:8080/rest/");
            }
        };
        Callable<ResponseEntity<String>> traceCallable1 = new TraceCallable<>
                (tracer, spanNamer, callable1, "get_service_1");

        ListenableFuture<ResponseEntity<String>> future1 = executor.submit(traceCallable1);

        ...

        // wait for results
        ResponseEntity<String> service1Response = future1.get();

        ...

        return serviceResponse.getBody();
    }

    private ResponseEntity<String> requestService(String serviceUrl) {
        return restTemplate.exchange(serviceUrl, HttpMethod.GET, null, String.class);
    }

}
```

Een `Tracer` en een `SpanNamer` worden geïnjecteerd door Sleuth en een `RestTemplate`

Figuur 3.4: Synchrone REST calls tussen microservices, gevisualiseerd in Zipkin



Figuur 3.5: Asynchrone REST calls tussen microservices, gevisualiseerd in Zipkin



door Spring. Omdat Sleuth automatisch tracing informatie toevoegt aan een `RestTemplate`, moet dit niet verder geconfigureerd worden.

3.2 Sampling strategieën en performantieverschillen

Er wordt onderzocht wat voor overhead het toevoegen van tracing informatie met zich meebrengt op de drie verschillende testopstellingen (zie 3.1.1). Omdat Sleuth gebaseerd is op Dapper, wordt bekeken hoe verschillende sampling strategieën de performantie beïnvloeden. Uit (Sigelman e.a., 2010) blijkt immers dat verzameling van traces de grootste invloed kan hebben op vertragingen in het netwerk bij het maken van requests.

Met een gesampelde trace wordt bedoeld dat de trace gemarkeerd is om verzameld te worden. In dit onderzoek betekent dit dat Sleuth een gesampelde trace zal doorsturen naar de Zipkin server. Om de sampling te definiëren voor een bepaalde service dient een Java bean aangemaakt te worden. Sleuth zal deze Sampler automatisch opmerken en gebruiken. Om bijvoorbeeld alle requests te tracen, kan gebruik gemaakt worden van Sleuth's `AlwaysSampler`.

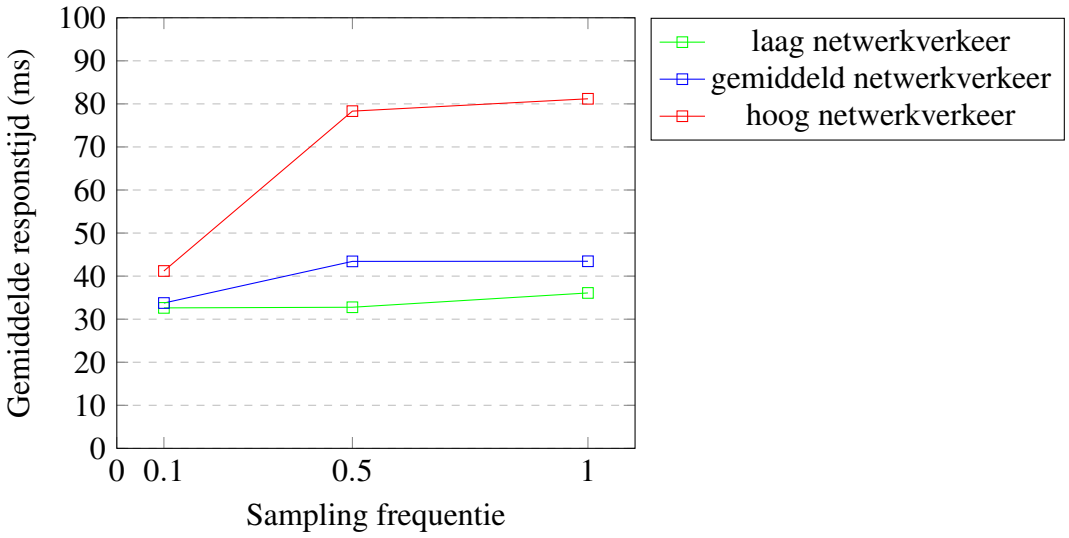
Om slechts een percentage van de requests te tracen, kan een eigen sampler gedefinieerd worden als volgt:

Listing 3.2: Sleuth percentage sampler

```
@Bean
public Sampler percentageSampler() {
    return new Sampler() {
        @Override
        public boolean isSampled(Span span) {
            Random rg = new Random();
            // trace 50% of all requests
            return rg.nextInt(10) > 4;
        }
    };
}
```

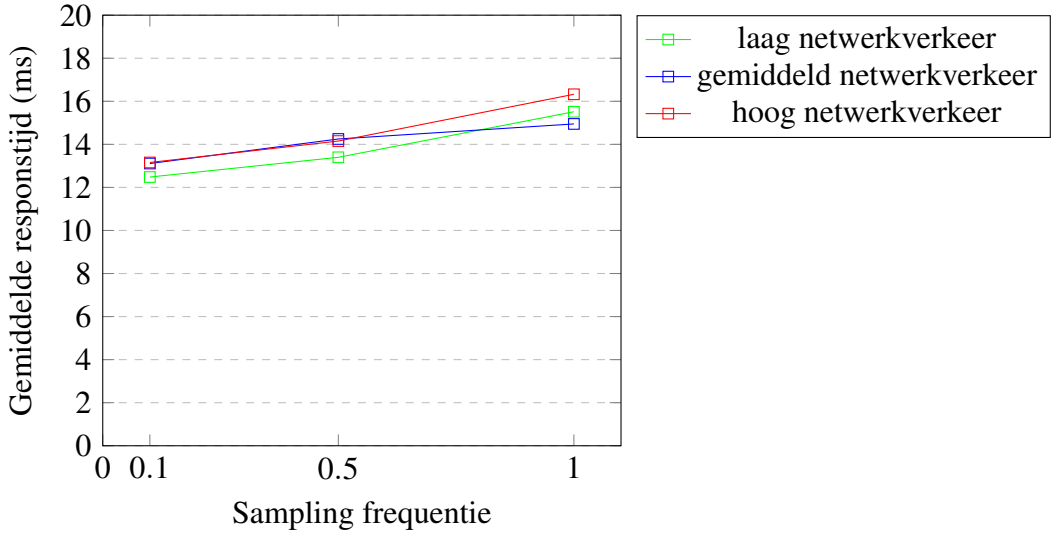
Dit onderzoek gaat na welke performantiegevolgen verschillende sampling percentages op de verschillende gedefinieerde microservice opstellingen (zie 3.1.1) hebben. Er wordt gebruik gemaakt van de Apache Benchmark (ab) tool om de responstijd te testen. Bij de testen wordt rekening gehouden met een *concurrency* variabele, dat aanduidt hoeveel requests er tegelijk worden uitgevoerd. Voor een hoog netwerkverkeer te simuleren worden er 1000 requests tegelijk uitgevoerd tot er 10 000 requests in totaal aangekomen zijn. De bovenlimiet 1000 is gekozen, omdat dit het maximaal aantal verbindingen is dat het gebruikte testsysteem tegelijk toelaat. Een gemiddeld netwerkverkeer wordt gesimuleerd door 200 requests tegelijk uit te voeren tot alweer 10 000 requests verwerkt zijn en een laag netwerkverkeer zal ten slotte 20 requests tegelijk uitvoeren.

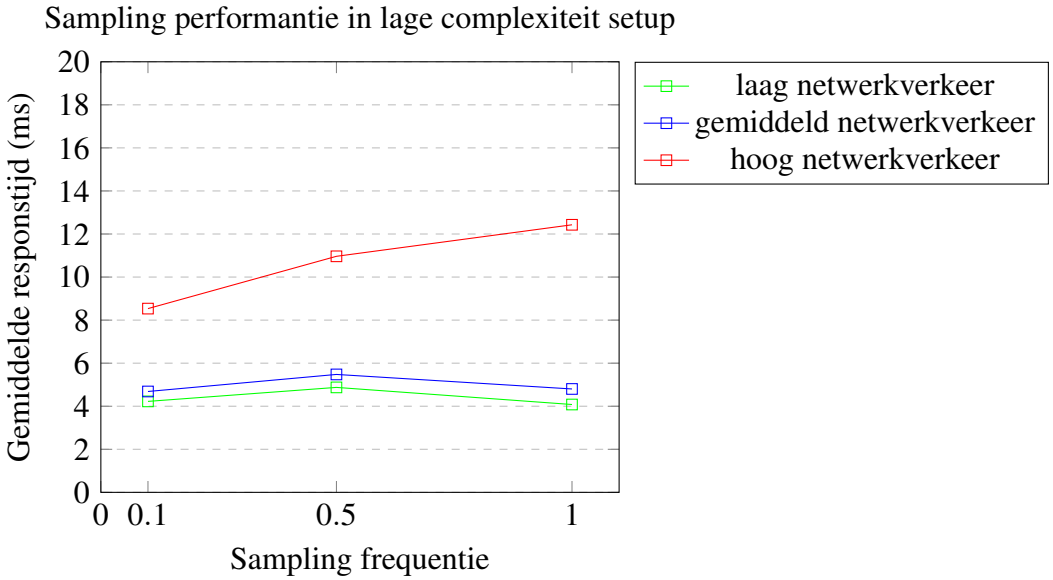
Sampling performantie in hoge complexiteit setup



Voor de hoge complexiteit setup blijkt dat tracing vanaf 50% sampling frequentie en bij hoog netwerkverkeer een grote stijging kent in responstijd ten opzichte van de responstijd bij 10% sampling. De responstijd stijgt van 41,2 ms naar 78,3 ms voor 50% sampling en 81,2 ms voor 100% sampling.

Sampling performantie in gemiddelde complexiteit setup





Voor de gemiddelde en lage complexiteit opstellingen heeft de sampling frequentie slechts een minimale impact op de totale performantie van het systeem.

3.3 Logging

Voor de logging worden er 4 extra microservices toegevoegd aan de testopstelling: Elastic-Search, Logstash, Fluentd en Kibana.

Om ervoor te zorgen dat de logs van ms-web en ms-service-n verzameld worden door Logstash of Fluentd wordt gebruik gemaakt van de Logback configuratie van Spring Boot, waar wordt aangegeven waar de Logstash of Fluentd server zich bevindt.

Listing 3.3: Voorbeeld Fluentd logback configuratie

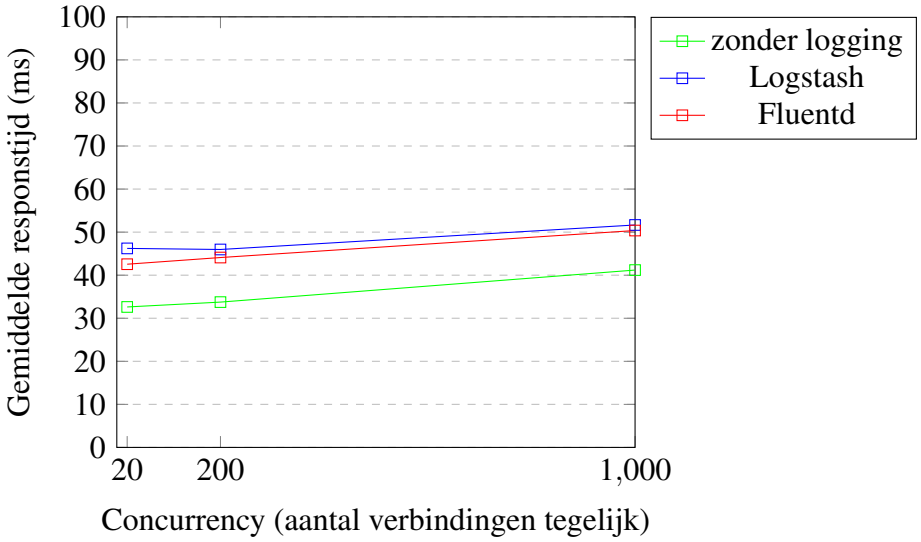
```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <property name="FLUENTD_HOST" value="${FLUENTD_HOST:-192.168.99.100}"/>
    <property name="FLUENTD_PORT" value="${FLUENTD_PORT:-24224}"/>
    <appender name="FLUENT" class="ch.qos.logback.more.appenders.DataFluentAppender">
        <tag>dab</tag>
        <label>normal</label>
        <remoteHost>${FLUENTD_HOST}</remoteHost>
        <port>${FLUENTD_PORT}</port>
    </appender>

    <logger name="com.frederic" level="INFO" additivity="false">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />
        <appender-ref ref="FLUENT" />
    </logger>

</configuration>
```

Voor Logstash dient de logstash-logback-encoder bibliotheek toegevoegd te worden aan elke microservice waarvan logging verzameld wilt worden. Voor Fluentd dienen de

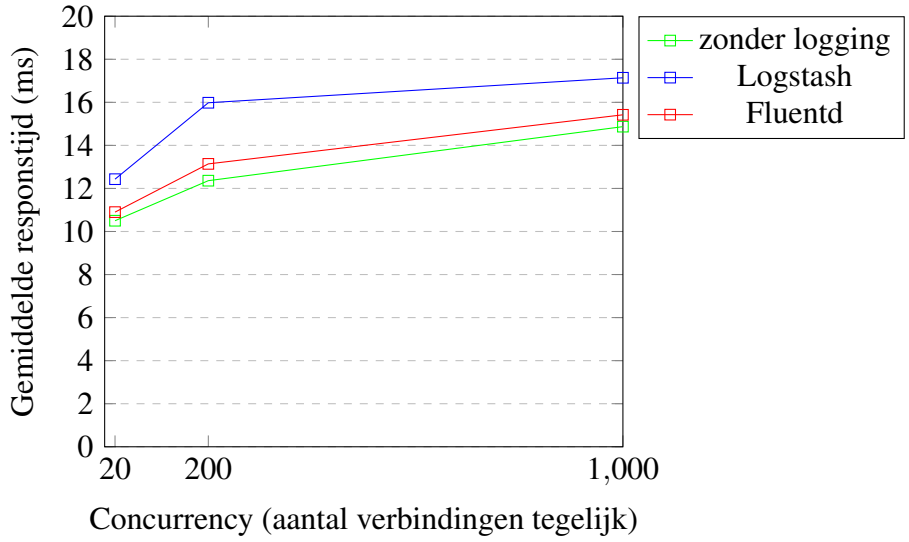
logback-more-appenders en fluent-logger bibliotheken toegevoegd te worden.

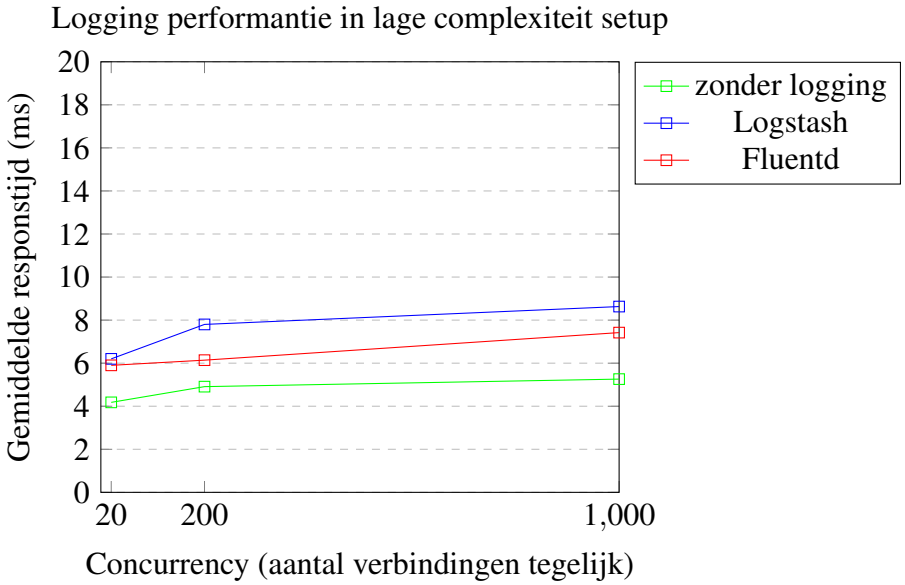
Er wordt onderzocht wat voor performantiegevolgen de verzameling van logs van de microservices naar ElasticSearch met zich meedraagt. De log shippers Logstash en Fluentd worden met elkaar vergeleken. Er wordt hier ook gebruik gemaakt van de ab tool om de responstijd te testen.

Logging performantie in hoge complexiteit setup



Logging performantie in gemiddelde complexiteit setup





In alle complexiteitsopstellingen worden slechts kleine performantiegevolgen genoteerd door het toevoegen van logging aggregatie. Fluentd kent over alle opstellingen heen een iets betere performantie dan Logstash, maar het verschil blijft miniem. Beide log shippers kennen een goede performantie bij een laag tot hoog netwerkverkeer.

4. Conclusie

Voor dit onderzoek was het onduidelijk wat de performantiegevolgen van tracing met Sleuth en Zipkin zouden zijn. Er kan nu gesteld worden dat tracing wel degelijk het systeem extra belast, maar dit enkel merkbaar zal zijn in een microservice architectuur die minstens zo complex is als in figuur 3.3. In zo een opstelling werd bij hoog netwerkverkeer (minstens 1000 tegelijke requests) een verdubbeling van de responstijd genoteerd. Belangrijk is dat die verdubbeling in responstijd pas optreedt vanaf een 50% sampling frequentie. Door het drukken van de sampling frequentie kan de performantielast, die tracing met zich meedraagt, geminimaliseerd worden. Bij een 10% sampling frequentie is de responstijd terug normaal. Dit ligt in de lijn met de verwachtingen voor dit onderzoek. In de Dapper paper (Sigelman e.a., 2010) werd immers genoteerd dat het niet nodig is om alle requests te tracen in een systeem dat gebukt gaat onder een hoog netwerkverkeer. Als er immers een probleem zou opduiken in het systeem, zal dat probleem zich herhalen in meerdere requests en uiteindelijk zal minstens één van die requests getraced worden.

Het was ook nog onduidelijk voor dit onderzoek of het verzamelen van de log berichten van de microservices extra performantiegevolgen hadden op het systeem. Er kan nu gesteld worden dat dit, in alle opstellingen met verschillend netwerkverkeer, geen grote invloed heeft op de totale performantie van het systeem. Opmerkelijk is dat Fluentd beter presteerde als Logstash in dit onderzoek. Er kan besloten worden dat, in een microservices opstelling met Spring Boot microservices, beter gebruik gemaakt kan worden van Fluentd in de plaats van Logstash.

Bijlagen

HoGent Bedrijf en Organisatie
Bachelorproef 2016 - 2017

Tracing in microservices met Spring Cloud Sleuth en Zipkin

Onderzoeksvoorstel Bachelorproef

Frederic Everaert[1], Geert Vandensteen[2]

Samenvatting
Microservices worden steeds populairder t.o.v. monolithische applicaties. De voordelen ervan worden vaak genoemd, maar een nadeel, dat vaak genegeerd wordt, is dat het opzetten van microservices complex kan zijn doordat requests verschillende onafhankelijke services doorkruisen. Gewone debugging tools zijn dan niet meer voldoende en een oplossing is nodig om de complexiteit te beantwoorden. Deze bachelorproef onderzoekt hoezeer tracing een oplossing is voor dit probleem. Specifiek Spring Cloud Sleuth en Zipkin worden onder de loep genomen waarbij de verschillende microservices opgezet worden met Docker. De verwachting is dat tracing een onmisbare tool is voor microservices en er hiervoor alleen maar meer en betere tools en oplossingen bedacht zullen worden in de toekomst.

Sleutelwoorden
Applicatieontwikkeling. Microservices — Tracing — Debugging

Contact: [1] frederic.everaert.u1028@student.hogent.be; [2] geert.vandensteen@synthetron.com

Inhoudsopgave

1. Introductie

Monolithische architecturen ruimen steeds meer baan voor microservice architecturen of kortweg microservices. Een monolitische applicatie is gebouwd als een enkelvoudige, zelfstandige eeheid. Bij een client-server model, kan bijvoorbeeld de applicatie langs de server zijde bestaan uit een enkele applicatie die de HTTP requests behandelt, logica uitvoert en data ophaalt of bijwerkt in de database. Het grote probleem van monolitische applicaties is dat ze moeilijk te onderhouden zijn. Een kleine verandering in een bepaalde functionaliteit van de applicatie kan ervoor zorgen dat andere delen van de applicatie ook bijgewerkt moeten worden. De volledige applicatie moet opnieuw gebuild en gedeployed worden. Als een bepaalde functionaliteit gescaled moet worden, moet de volledige applicatie gescaled worden. Microservices bieden hiervoor een oplossing, aangezien de verschillende functionaliteiten elk een op zich zelf staande service kunnen vormen.

Bij microservices schieten echter de traditionele tools voor debugging tekort. Een enkele service kan niet het volledige beeld geven over bijvoorbeeld de performantie van de applicatie in zijn geheel. Om dit te verhelpen, kunnen requests getraced worden. Een trace is de volledige reisweg van een request die spans bevat voor alle doorkruiste microservices. Een span bestaat uit tags of metadata zoals de start- en stoptijdstippen. Deze data kan dan verzameld worden om een volledig beeld van het gedrag van de applicatie te geven.

De bedoeling van dit onderzoek is om de meerwaarde van tracing in microservices aan te tonen en ook om eventuele tekortkomingen vast te stellen. Er wordt specifiek gekeken naar tracing van requests met behulp van Spring Cloud Sleuth en Zipkin. In de praktijk gaat men niet alle requests gaan tracen om onnodige overhead te vermijden. Beslissen welke requests getraced worden en welke niet wordt sampling genoemd. Wat kunnen bijvoorbeeld enkele interessante use cases zijn waarbij er op een intelligente manier gesampled kan worden? Ten slotte visualiseert Zipkin enkel de verschillende traces, maar niet de log berichten, deze worden dan ook nog niet verzameld. Om logs te verzamelen en visualiseren wordt gekeken naar ElasticSearch, Logstash en Kibana. Er wordt aangetoond hoe deze tools in combinatie met Zipkin helpen om de complexiteit van microservices in bedwang te houden.

2. State-of-the-art

Onderzoeken rond distributed tracing systemen zijn schaars, maar de technologieën die gebruikt worden in dit onderzoek, namelijk Spring Cloud Sleuth en Zipkin, zijn gebaseerd op Google Dapper. (Sigelman e.a., 2010) Deze technische paper

beschrijft in detail de werking van Google's distributed tracing systeem. Net zoals Dapper gebruikt Sleuth een annotatie gebaseerde methode om tracing toe te voegen aan requests. De terminologie is volledig overgenomen. Een trace bevat meerdere spans voor elke hop naar een andere service. Spans die dezelfde trace id bevatten, maken deel uit van dezelfde trace. Een groot ontwerpdoel van Dapper was om de overhead voor tracing zo laag mogelijk te houden. Sampling werd hierdoor geïntroduceerd. In een eerste productie gebruikten ze eenzelfde sampling percentage voor alle processen bij Google, gemiddeld één trace per 1024 kandidaten. Dit schema bleek heel effectief te zijn voor diensten met veel verkeer, maar bij diensten met minder verkeer gingen er zo interessante events verloren. Als oplossing werd toegelaten dat dit sampling percentage wel aan te passen was, ook al wou Google met Dapper dit soort manuele interventie vermijden. Spring Cloud Sleuth neemt dezelfde instelling over en laat ook toe om zelf sampling in te stellen.

3. Methodologie

Dit onderzoek stelt een microservices architectuur op door gebruik te maken van Spring Boot en Docker. Elke microservice is een Spring Boot applicatie die draait in zijn eigen Docker container. Met Docker Compose worden de verschillende containers opgestart. Voor de tracing wordt gebruik gemaakt van Spring Cloud Sleuth. Er wordt uitgelegd wat traces precies zijn en hoe Sleuth requests volgt doorheen de microservices. Om de data te verzamelen en te visualiseren wordt Zipkin gebruikt. Voor log aggregatie en visualisatie wordt gebruik gemaakt van ElasticSearch, Logstash en Kibana.

Er zullen performantietesten worden uitgevoerd om uit te zoeken hoezeer het systeem belast wordt door tracing toe te voegen aan requests. Een succesvolle sampling strategie zou niet voor merkbare vertraging mogen zorgen ($< 2\%$ vertraging). Eenvoudige sampling is percentueel gewijs te onderzoeken: wat is de vertraging als er $\frac{1}{1}$ tracing wordt toegevoegd, $\frac{1}{2}$, $\frac{1}{4}$... Maar dit hangt dan ook af van situatie tot situatie. Voor een drukke webservice met veel requests per seconde en veel verschillende microservices zal een andere sampling strategie nodig zijn dan een eenvoudigere setup die niet veel requests per seconde ontvangt. Er worden ook complexere strategieën gedefinieerd qua sampling. In plaats van percentueel, enkel alle 500 requests tracen bijvoorbeeld. Er worden zo een aantal use cases uitgezocht en op die manier worden de gevolgen aangetoond van een goede sampling strategie op basis van situatie.

4. Verwachte resultaten

Voor de percentuele sampling wordt verwacht dat hoe hoger de sampling frequentie, hoe zwaarder het systeem belast zal worden. Bij andere situaties zal een complexere sampling strategie voordeliger zijn in vergelijking met een percentuele sampling. Verder wordt verwacht dat de visualisatie van tracing en logging de meerwaarde van Zipkin en Kibana zou schetsen en de voor- en nadelen van tracing aangetoond worden in verschillende situaties.

5. Verwachte conclusies

De verwachte conclusie is dat tracing, indien goed toegepast, een meerwaarde is voor microservices en dit gestaafd is met uitgewerkte situaties in een opgestelde testomgeving.

Referenties

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., … Shanbhag, C. (2010). *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Google, Inc. Verkregen van http://research.google.com/archive/papers/dapper-2010-1.pdf

Bibliografie

Docker. (2016). What is Docker? Verkregen 12 januari 2017, van https://www.docker.com/what-docker

Docker Inc. (2015). Overview of Docker Compose, 1–6. Verkregen van https://docs.docker.com/compose/overview/

OpenZipkin · A distributed tracing system. (g.d.). Verkregen 12 januari 2017, van http://zipkin.io/

Pivotal Software, I. (2014). Spring Framework. Verkregen van https://projects.spring.io/spring-framework/

Pivotal Software, I. (2015a). Spring Boot. Verkregen van https://projects.spring.io/spring-boot/

Pivotal Software, I. (2015b). Spring Cloud. Verkregen 28 december 2016, van http://projects.spring.io/spring-cloud/

Sigelman, B. H., Andr, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., … Shanbhag, C. (2010). Dapper , a Large-Scale Distributed Systems Tracing Infrastructure. *Google Research*, (April), 14. doi:dapper-2010-1

Lijst van figuren

Lijst van tabellen