한국정보보호학회
Korea Institute of Information Security & Cryptology

2020년 한국정보보호학회 동계학술대회

CISC-W'20

Conference on Information Security and Cryptography-Winter 2020

일자 2020년 11월 28일 (토) **장소** 온라인 컨퍼런스

접속 및 시청방법 등록자에 한하여 개별 공지

Proceedings

주최 한국정보보호학회 Korea Institute of Information Security & Cryptology **주관** 서울대학교 SEOUL NATIONAL UNIVERSITY

후원 국가정보원 National Intelligence Service Korea, 과학기술정보통신부, 행정안전부, KISA 한국인터넷진흥원, ETRI 한국전자통신연구원 Electronics and Telecommunications Research Institute, NSR 국가보안기술연구소, cen 아이티센 ITCen

분산컴퓨팅 환경에서 효율적인 분산 트랜잭션 추적을 위한 샘플링 오버헤드 연구

이병용*, 최상훈*, 박기웅†

*세종대학교 시스템 보안 연구실, † 세종대학교 정보보호학과

A Study on Sampling Overhead for Efficient Distributed Transaction Tracking in Distributed Computing Environments

Byeong-Yong Yi*, Sang-Hoon Choi*, Ki-Woong Park†

*SysCore Lab., Sejong University.

† Department of Computer and Information Security, Sejong University

요 약

최근 정보통신 분야에서 클라우드 서비스의 편리성과 비용적 측면의 장점을 주목하면서, 이를 활용하기 위해 마이크로서비스 아키텍처(Microservice Architecture)형태의 서비스를 적극적으로 사용하고 있다. 하지만, 마이크로서비스 아키텍처는 모놀리식(Monolithic) 환경보다 복잡한 서비스 구조로 되어 있는 특징이 있다. 마이크로서비스 아키텍처에서 서비스를 개발 및 유지보수하는 과정에선 테스트 및 디버깅 과정이 필요하므로, 서비스 요청으로 인해 발생하는 이벤트를 기록하는 분산추적(Distributed Tracing)기술이 지속 연구되고 있다. 분산추적은 여러 가지 방법이 있으나, 최근에는 이벤트 간 인과관계 파악에 유용한 콘텍스트 전파 방법을 대다수의 추적 도구가 사용하고 있다. 이 과정에선, 추적을 위해 서비스의 소스 코드를 수정할 필요가 있으며, 추적 데이터를 일부 저장하는 과정이 필요하다. 하지만, 이러한 과정은 오버헤드를 발생시킨다는 문제점을 가지고 있다. 본 논문은 오픈 소스 분산추적 도구인 Jaeger에서 샘플링 과정이 유발하는 오버헤드를 측정하고, 주요 오버헤드 발생 요인을 분석한다. 결론으로 본 논문에서 분석한 오버헤드 발생 요인 정보를 근거로 한 추후 연구에 관해서 서술한다. 본 논문은 분산추적 도구의 성능향상에 활용되어, 마이크로서비스 아키텍처에서 작동하는 서비스의 성능 개선에 더욱 힘쓸 수 있을 것으로 예상한다.

I. 서론

모놀리식 아키텍처에서 마이크로서비스 아키텍처로의 서비스 구조 변화는 최근 빠르게 발전하고 있는 클라우드 컴퓨팅 기술에 힘입어 급속도로 진행되고 있다. O'REILLY 사에서 메일링 리스트의 등록된 독자들을 대상으로 진행한 마이크로서비스 아키텍처 이용 동향에 관련된 설문조사 살펴보면, 응답자의 61%는 마이크로서비스 아키텍처를 1년 이상 사용 중이라고 응답하였다[1]. 마이크로서비스 아키텍처는 기존의 모놀리식 환경에서 수행하던 세부 기능들을 모듈 형태로 나누면서, 서비스의 배포 및 구동 과정이 매우 유연하고 특정 환경에 종속되지 않는 특징을 갖고 있다[2]. 이러한 소프트웨어 구조는 문제 발생 시 해당 문제를 복구하는 장애 극복기능(Failover), 과도한 트래픽 발생 시, 트래픽을 분산함으로써 가용성을 확보하는 로드 밸런싱 기능(Load Balancing), 새로운 패치를 중단없이 배포하는 지속 배포 기능(Continuous Deployment) 등의 장점이 있다.

하지만, 마이크로서비스 아키텍처는 장점과

† 교신저자: 박기웅 (세종대학교 정보보호학과 교수)

본 연구는 과학기술정보통신부의 재원으로 정보통신기획평가원(IITP)의 지원(Project No. 2018-0-00420)을 받아 수행된 연구임.

함께 그에 따른 한계점도 가지고 있다. 내부 기능이 작고 세분된 마이크로서비스 아키텍처의 특징은 전체 서비스 구조를 매우 복잡하게 만든다. 서비스 구조가 복잡하면 문제 발생 시, 근본적인 원인 분석이 어려워지고, 결국 서비스 운영을 어렵게 한다[3]. 앞서 예를 든 O'REILLY 사의 동일 설문조사에선 전체 응답자의 92%에 달하는 응답자가 마이크로서비스로 아키텍처로의 소프트웨어 구조 변경을 대체로 긍정적으로 여겼으나, 마이크로서비스 아키텍처 사용 시 겪는 고충으로 시스템 복잡도와 관련된 답변이 56%에 달할 정도로 높은 비율을 보였다.

마이크로서비스 아키텍처는 구조적으로 복잡할 뿐만 아니라, 요청 수행 과정이 gRPC[5], REST[6]와 같은 기술을 활용하여 물리적으로 떨어진 호스트의 프로세스와 통신하며 실행되는 분산 동시성[7]을 기반으로 작동한다. 마이크로서비스 아키텍처에서도 서비스 개발 단계에서 테스트와 디버깅이 필요하지만, 모놀리식 아키텍처에서 사용하던 소프트웨어 추적 도구로는 의미 있는 데이터를 획득하는 데 한계를 가지고 있다. 이러한 한계점을 해결하기 위해서, 마이크로서비스 환경에 적합한 분산추적(Distributed Tracing)기법이 연구되기 시작하였다.

분산추적 기법은 요청 간 인과관계 정보를 제공하는 것이 핵심이므로, 비교적 정확한 인과관계 정보를 제공하는 콘텍스트 전파(Context Propagation)을 사용한다. 하지만, 콘텍스트 전파 기법은 추적을 수행하는 과정에서 샘플링이 필요하며, 이 과정은 오버헤드를 발생시킨다는 문제점을 가지고 있다. 샘플링은 추적 데이터 저장 비율을 조절함으로써 오버헤드를 조절할 수 있지만, 추적 데이터의 정확도가 떨어지는 단점이 있다. 즉, 샘플링 비율을 낮게 설정하면 추적 데이터가 불완전해지는 문제가 발생하며, 샘플링 비율을 높게 설정하면 과도한 오버헤드가 발생한다. 따라서, 샘플링 과정 자체의 오버헤드를 근본적으로 낮출 필요성이 요구된다.

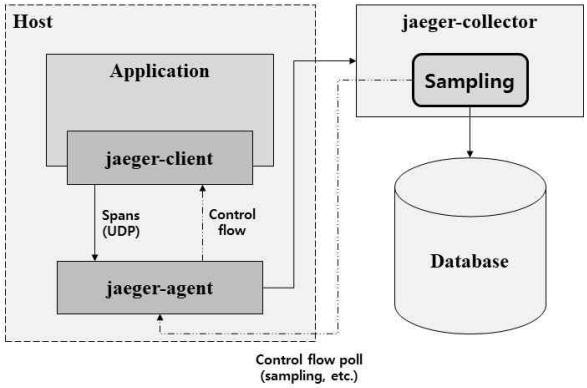
본 논문은 오픈 소스 분산추적 도구인 Jaeger의 샘플링 과정에서 발생하는 오버헤드에 대해서 분석을 수행한다. 본 논문의 구성은 다음과 같다. 2장에선 분산추적과 관련된 연구를 소개한다. 3장에선 분산추적의 샘플링 과정에서 발생하는 오버헤드를 분석한다. 4장에선 결론을 내고, 추후 연구에 관해서 서술한다.

II. 관련 연구 및 배경지식

2장에선 마이크로서비스 아키텍처에서 분산추적을 수행하기 위해 공개된 관련 연구와 도구에 대해서 살펴본다.

2.1 추적 기법 관련 연구

분산추적에 관한 연구는 Barham et al. 연구진이 발표한 Magpie[8]를 비롯하여 이전부터 활발하게 진행되어왔으나, 현재 주로 사용 중인 기법은 Sigelman et al[9]. 연구진에 의해서 제안되었다. 해당 연구에서 제안된 추적 기법은 프로토콜 헤더에 추적에 관한 정보를 추가하는 콘텍스트 전파기법을 사용하였으며, 추적 데이터를 Trace와 Span 개념을 사용하여 분류하였다. Trace는 특정 요청에 대한 작업을 수행하기 위해 진행된 일련의 전체 과정이며, Span은 Trace에서 서비스를 거쳐 수행하는 작은 단위의 작업이다. Trace와 Span을 활용하여 분산 트랜잭션을 추적하기 위해선, 추적하고자 하는 마이크로서비스에 추적에 관한 소스 코드를 삽입하는 것이 필요하다. 요청이 해당 마이크로서비스에 도달하면, 시간 정보(Timestamp)가 각 마이크로서비스의 로컬 호스트 머신에 기록되고, 이러한 데이터는 주기적으로 추적 데이터 전송 데몬에 의해서 수집되고 전송된다. 모든 분산 트랜잭션 데이터를 이러한 방식으로 수집하면 수집 백엔드에 매우 큰 연산 오버헤드가 발생하며, 추적 데이터를 저장을 위한 거대한 저장공간이 필요하므로, Dapper는 전체 추적 데이터 중 일부의 데이터만을 기록하는 샘플링 방식을 도입하였다. Dapper의 샘플링 비율은 사용자의 설정에 따라서 조절할 수 있으며, 기본적으로 1024개의 Trace 중 임의의 Trace를 기록하는 확률적 샘플링을 사용하고 있다. 확률적



[그림 1] Jaeger의 기본적인 샘플링 구조

샘플링을 사용하면, 수집 초기에는 부분적인 데이터만을 획득할 수 있으나, 추적 데이터를 데이터베이스에 쌓음으로써 전체 통신 흐름을 알 수 있는 추적 데이터로 사용할 수 있다. 하지만, 샘플링 기법을 사용하더라도, 오버헤드 문제는 언제든지 발생할 수 있다. Dapper 논문에서 발표한 실험 자료에 따르면, 샘플링 비율에 따라서, 평균 지연시간의 최대 16%까지 차이가 나는 것을 확인할 수 있었다. 즉, 이러한 추적 기법은 비교적 정확한 인과관계 정보를 제공하는 장점이 있으나, 추적 정확도와 시스템 오버헤드 사이에서 사용자의 선택을 요구한다.

Dapper 논문에서 사용하는 추적 기법을 기반으로 하는 상용 분산추적 도구로는 구글의 Operations(Stackdriver)[10], 아마존의 X-ray[11] 등이 있으며, 오픈 소스 분산추적 도구로는 OpenZipkin[12], Jaeger[13] 등이 있다.

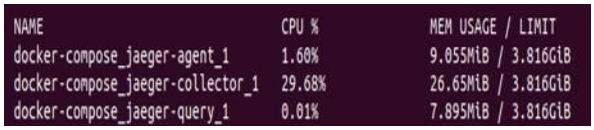
2.2 Jaeger 추적 도구 배경지식

Jaeger는 Uber Technology에서 공개하였으며, 현재는 CNCF(Cloud Native Computing Foundation)에서 호스팅 중인 오픈 소스 분산추적 도구이다. Jaeger는 Dapper 분산추적의 핵심 개념인 Span과 Trace를 개념을 사용하고 있으며, 샘플링 기반의 분산추적 자료수집을 수행한다. Jaeger의 주요 구성요소는 [그림 1][14]과 같이 jaeger-client, jaeger-agent, jaeger-collector로 구성되어있으며, 이외에도 추적 데이터를 사용자에게 제공하는 jaeger-query, jaeger-collector로부터 추적 데이터를 전송받아 메시지 큐에 전달하는 jaeger-ingester 등이 있다.

jaeger-client는 서비스에 삽입된 소스 코드에서 생성된 추적 데이터를 UDP를 사용하여 jaeger-agent에 전송한다. jaeger-agent는 전송받은 네트워크 데이터를 jaeger-collector에 전송하는 네트워크 데몬이다. 또한, 추적 대상인 모든 호스트에서 작동하는 특징을 가지고 있다. 샘플링 기능을 수행하는 소스 코드는 jaeger-collector에 구현되어 있으며, 데이터베이스와의 상호작용을 통해 추적 데이터를 저장한다. Jaeger는 플러그인을 통해 추적 데이터를 데이터베이스에 적합한 형태로 변환한다. 현재 데이터베이스 플러그인은 Cassandra, Elasticsearch를 지원하고 있으며, Kafka와 같은 메시지 큐에 데이터를 전송하는 상황을 가정하여, Kafka 플러그인 또한 지원한다. Jaeger는 sampling_strategies.json 파일을 통해 샘플링 기법, 샘플링 비율을 설정할 수 있다. 샘플링 비율은 0에서 1 사이의 숫자에서 설정할 수 있으며, 0일 경우 샘플링을 수행하지 않고, 1일 경우 모든 추적 데이터를 샘플링한다. 샘플링 비율을 1에 가깝게 설정하면, 만에 하나 발생할 수 있는 비정상적인 통신까지 탐지할 수 있는 확률이 높아지지만, 급격하게 오버헤드가 상승하며 다수의 저장공간이 필요하다는 단점을 가지고 있다.

III. 샘플링 오버헤드 분석

3장에선 소프트웨어 프로파일링 기법을 활용하여, Jaeger 추적 도구의 샘플링 과정에서 발생하는 오버헤드를 분석한다. 오버헤드 분석을 위해 Go 언어를 사용하여 간단한 서버-클라이언트 통신 환경을 구성하였으며, 서비스의 함수마다 추적을 수행하기 위한 소스 코드를 추가하였다. 해당 서버-클라이언트는 특정 값의 존재 여부에 대한 요청을 보내면, 이를 데이터베이스에서 찾아서 요청에 알맞은 응답을 수행하는 구조이다. 추적을 진행하기 위해서 기본적인 Jaeger 백엔드(jaeger-agent, jaeger-collector, jaeger-query)와 추적 데이터 저장을 위한 데이터베이스는 도커 컨테이너를 사용하여 실행하였다. Jaeger 백엔드를 구동시킨 상황에서, 원격지로부터 서버에 지속적인 요청을 보냈을 때,



NAME	CPU %	MEM USAGE / LIMIT
docker-compose_jaeger-agent_1	1.60%	9.055MiB / 3.816GiB
docker-compose_jaeger-collector_1	29.68%	26.65MiB / 3.816GiB
docker-compose_jaeger-query_1	0.01%	7.895MiB / 3.816GiB

[그림 2] Jaeger 백엔드의 오버헤드 측정

백엔드 구성요소 중 어느 부분에 가장 많은 부하가 발생하는지 확인하였으며, 그 결과는 [그림 2]와 같다. Jaeger 백엔드를 도커 컨테이너로 구성하였기 때문에, 'docker stats' 명령어를 사용하여 부하를 측정하였다. 외부 데이터베이스를 제외하면, Jaeger의 백엔드 구성 요소 중에선 docker-compose_jaeger-collector_1(jaeger-collector)가 가장 많은 부하를 발생시켰다. 추가로 상세한 부하 발생 요인을 분석하기 위해 리눅스 성능 분석 도구인 perf를 사용하여 jaeger-collector를 대상으로, 샘플링 수행 시 가장 많이 호출되며, 오버헤드를 발생시키는 함수에 대한 분석을 수행하였다. 대부분의 오버헤드는 사용자가 Span데이터를 구분하기 쉽도록 인덱스를 추가하는 writeIndexes() 호출 과정에서 많은 오버헤드가 발생하였다.

IV. 결론

본 논문은 마이크로서비스 아키텍처에서 디버깅 및 테스트를 위한 분산추적의 샘플링 오버헤드를 낮추기 위해, 오픈 소스 분산추적 도구로 활발히 사용되는 Jaeger의 오버헤드에 관한 연구를 수행하였다. Jaeger는 샘플링 과정 중에서도 인덱스를 작성하는 과정에서 가장많은 오버헤드를 발생시키는 것을 확인할 수 있었다. 추후 연구에선 오버헤드 발생 원인을 해결하고, 실제 서비스환경에서도 활용 가능한 샘플링 방안을 설계 및 구현할 예정이다.

[참고문헌]

[1] Microservices Adoption in 2020. https://www.oreilly.com/radar/microservices-adoption-in-2020/

[2] Microservices. https://martinfowler.com/articles/microservices.html

[3] Kitajima, Shinya, and Naoki Matsuoka. "Inferring calling relationship based on external observation for microservice architecture." International Conference on Service-Oriented Computing. Springer, Cham, 2017.

[4] Kitajima, Shinya, and Naoki Matsuoka. "Inferring calling relationship based on external observation for microservice architecture." International Conference on Service-Oriented Computing. Springer, Cham, 2017.

[5] gRPC. https://grpc.io/

[6] Fielding, Roy T., and Richard N. Taylor. Architectural styles and the design of network-based software architectures. Vol. 7. Irvine: University of California, Irvine, 2000.

[7] Bernstein, Philip A., and Nathan Goodman. "Concurrency control in distributed database systems." ACM Computing Surveys (CSUR) 13.2 (1981): 185-221.

[8] Barham, Paul, et al. "Using Magpie for request extraction and workload modelling." OSDI. Vol. 4. 2004.

[9] Sigelman, Benjamin H., et al. "Dapper, a large-scale distributed systems tracing infrastructure." (2010).

[10] Operations. https://cloud.google.com/products/operations?hl=ko

[11] X-ray. https://aws.amazon.com/ko/xray/

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

[13] Jaeger. https://www.jaegertracing.io/

[14] Jaeger Architecture. https://www.jaegertracing.io/docs/1.13/architecture/