PaaS 클라우드 컴퓨팅을 위한 컨테이너 친화적인 파일 시스템 이벤트 탐지 시스템

Container-Friendly File System Event Detection System for PaaS Cloud Computing

전우진, 박기웅[1)]
Woo-Jin Jeon, Ki-Woong Park

(05006) 서울특별시 광진구 능동로 209 세종대학교
정보보호학과 시스템보안 연구실, 정보보호학과
woojinjeon929@gmail.com, woongbak@sejong.ac.kr

요 약

최근 컨테이너 기반의 PaaS (Platform-as-a-Service)를 구축하는 트렌드가 확대되고 있다. 컨테이너 기반 플랫폼 기술은 클라우드 컴퓨팅을 구축하기 위한 하나의 주요 기술로써, 컨테이너는 가상 머신에 비해 구동 오버헤드가 적다는 장점이 있어 수백, 수천 대의 컨테이너가 한 대의 물리적 머신에서 구동될 수 있다. 하지만 이러한 클라우드 컴퓨팅 환경에서 구동되는 다수의 컨테이너에 대한 스토리지 로그를 기록하고 모니터링하는 것은 상당한 오버헤드가 발생한다. 따라서 본 논문에서는 클라우드 컴퓨팅 환경에서 구동되는 컨테이너의 파일 시스템 변경 이벤트를 탐지할 때 발생하는 두 가지 문제점을 도출하고, 이를 해결하여 PaaS 형태의 클라우드 컴퓨팅 환경에서 컨테이너 파일 시스템 이벤트 탐지를 위한 시스템을 제안하였다. 성능 평가에서는 본 논문에서 제안한 시스템의 성능에 대한 세 가지 실험을 수행하였고, 본 논문에서 제안한 모니터링 시스템은 컨테이너의 CPU, 메모리 읽기 및 쓰기, 디스크 읽기 및 쓰기 속도에 아주 미세한 영향만을 발생시킨다는 것이 실험을 통해 도출되었다.

Abstract

Recently, the trend of building container-based PaaS (Platform-as-a-Service) is expanding. Container-based platform technology has been a core technology for realizing a PaaS. Containers have lower operating overhead than virtual machines, so hundreds or thousands of containers can be run on a single physical machine. However, recording and monitoring the storage logs for a large number of containers running in cloud computing environment occurs significant overhead. This work has identified two problems that occur when detecting a file system change event of a

※ 본 연구는 한국연구재단 지원사업(2017R1C1B2003957) 및 2018년도 과학기술정보통신부의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2018-0-00420, API 호출 단위 자원 할당 및 사용량 계량이 가능한 서버리스 클라우드 컴퓨팅 기술 개발)

1) 교신저자

논문투고일자 : 2019. 02. 08 심사일자 : 2019. 02. 08 게재확정일자 : 2019. 02. 22

container running in a cloud computing environment. This work also proposes a system for container file system event detection in the environment by solving the problem. In the performance evaluation, this work performed three experiments on the performance of the proposed system. It has been experimentally proved that the proposed monitoring system has only a very small effect on the CPU, memory read and write, and disk read and write speeds of the container.

키워드: 클라우드 컴퓨팅, 컨테이너, inotify, 파일 시스템, 이벤트 탐지
Keyword: Cloud Computing, Container, inotify, File System, Event Detection

1. 서론

2018년 11월 22일 오전, 클라우드 시장 점유율 1위인 미국 아마존 웹서비스 (Amazon Web Services) [1] 서버가 다운되었다 [2]. 이로 인해 해당 서비스를 사용하고 있는 업체들의 온라인 사이트와 모바일 앱은 마비되었고, 기업의 매출에 피해를 발생시켰다. 또한 전 세계 최대 동영상 서비스를 지원하고 있는 구글의 유튜브 (YouTube) [3]에서도 2018년 10월 17일 전 세계 동시 서비스 장애를 일으켰다 [4]. 이와 같은 사고 발생 시 스토리지에 저장되어 있는 기업 기밀 또는 사용자 개인정보와 같은 민감한 데이터에 대해 안전하게 보안하는 것이 매우 중요하다. 또한 사고가 악의적인 목적을 가진 공격자에 의한 해킹인지, 단순 서버 장애인지에 대하여 명확한 원인 규명이 필요하다. 따라서 중요한 데이터가 저장되는 스토리지에서 발생되는 모든 로그에 대하여 정확하게 기록하여 확인하는 스토리지 모니터링은 필수적이다 [5, 6]. 이는 최근 DevOps (Development + Operations) [7, 8]가 활성화됨에 따라, PaaS (Platform-as-a-Service) [9] 기반의 컨테이너 [10, 11] 플랫폼 기술을 클라우드 컴퓨팅 환경에 적용하여 클라우드 서비스를 구축하는 트렌드가 확대되어 컨테이너 기반의 스토리지 로그 기록의 중요성 또한 더욱 강조되고 있다. 컨테이너 기술은 컨테이너 애플리케이션을 제외한 모든 호스트 자원을 공유하여 사용하기 때문에 가상 머신에 비해 구동 오버헤드가 적다는 장점이 있어, 수백, 수천 개의 컨테이너가 한 대의 물리적 서버에서 구동될 수 있다 [12]. 이에 따라 클라우드 컴퓨팅 환경에서 구동되고 있는 수백, 수천 개의 컨테이너에 대한 각각의 스토리지 로그를 기록하고 모니터링하는 것은 보안적 관점에 있어 매우 중요하며, 동시에 수백, 수천 개의 컨테이너를 모니터링하는 것은 용량적, 연산적 관점에 있어 상당한 시스템 오버헤드가 발생되기 때문에 매우 어려운 일이다.

따라서 본 논문에서는 클라우드 컴퓨팅 환경에서 구동되는 다수의 컨테이너를 모니터링 할 때 발생하는 두 가지 문제점을 도출하고, 이를 해결하여 PaaS 클라우드 컴퓨팅 환경을 위한 컨테이너 친화적인 파일 시스템 변경 이벤트 탐지 시스템을 제안한다. 첫 번째 문제점은 클라우드 컴퓨팅 환경에서 구동되는 수백, 수천 개의 컨테이너를 모니터링하는 것은 구동, 정지 및 종료 시점이 각기 다른 다수의 컨테이너 상태에 있어, 컨테이너가 정지되어 구동되지 않는 상태에도 모니터링이 수행되기 때문에 모니터링 시스템 성능에 있어서 상당한 오버헤드가 발생하는 것이다. 또한 일반적인 모니터링 프로세스는 모니터링 대상 컨테이너가 구동된 이후 대상을 인지하고 모니터링을 실행하여 이벤트를 탐지하는 순서로 수행되기 때문에 컨테이너 구동 이후부터 모니터링이 수행되기 전까지 모니터링이 수행되지 않는 음영 영역이 발생하여 모니터링 정확성이 떨어진다. 이러한 음영 영역이 발생하게 되면 모니터링 대상 컨테이너의 구동부터 종료 시점의 모든 것을 모니터링 한다고 보기 어렵다. 따라서 이와 같은 문제점을 해결하기 위해 Docker 컨테이너 [18]의 라이프 사이클을 이해하고, 이를 기반으로 컨테

이너가 구동, 정지 및 종료 시점을 인지함으로써 컨테이너가 구동되는 중에만 모니터링이 수행될 수 있도록 설계 및 구현하였다. 이에 따라 컨테이너가 구동되지 않는 모니터링이 불필요한 상황에서는 모니터링이 수행되지 않도록 구현하여 모니터링의 시스템 성능 효율을 높였다. 또한 컨테이너에 특화된 무손실 모니터링 알고리즘을 고안하여 컨테이너 구동 후 모니터링 실행 전까지 모니터링이 수행되지 않는 음영 영역이 발생하지 않도록 하는 무손실 모니터링 알고리즘을 구현하여 모니터링의 정확성을 높이도록 하였다. 두 번째 문제점은 기존의 파일 시스템 변화를 탐지하는 데 주로 사용되었던 inotify 도구에서 나타났다. inotify는 모니터링 하는 대상 디렉터리의 하위 디렉터리에 대한 파일 시스템 변경 이벤트는 탐지하지 못한다는 한계점을 가지고 있었다. 이와 같은 한계점은 첫 번째 문제점을 해결하여 모니터링에 대한 시스템 성능 효율과 정확성을 높였다고 하더라도 모니터링 탐지 성능 자체에 대한 정확성을 낮추기 때문에 제대로 된 모니터링을 수행한다고 할 수 없다. 따라서 본 논문에서는 inotify의 한계점을 Docker 컨테이너의 OverlayFS(Overlay File System) [19, 20, 21] 드라이버 구조를 분석하여 시스템 성능 효율적으로 하위 디렉터리의 파일 시스템 변경 이벤트를 모니터링 가능하도록 하는 메커니즘을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 연구와 관련된 스토리지 이벤트 탐지 연구와 클라우드 모니터링 연구에 대하여 설명하고 기존 연구들의 한계점을 도출한다. 3장에서는 본 논문에서 제안한 PaaS 클라우드 컴퓨팅 환경에서의 컨테이너 친화적인 파일 시스템 변경 이벤트 탐지 시스템의 설계 및 구현에 대하여 기술한다. 4장에서는 제안된 알고리즘의 성능 평가와 시스템 오버헤드를 측정하고, 그 결과에 대해 기술한다. 마지막 5장에서는 결론에 대해 기술한다.

2. 관련 연구

본 장에서는 논문에서 제안하는 파일 시스템 변경 이벤트 탐지 기술과 클라우드 컴퓨팅 환경 기반의 이벤트 모니터링 기술에 관련된 기존 연구 결과를 설명하고 이들 연구가 가진 한계점을 도출한다.

2.1 파일 시스템 변경 이벤트 탐지 연구

2.1.1 dnotify

dnotify는 fcntl 호출의 하위 기능 중 하나인 리눅스 커널에 대한 파일 시스템 이벤트 모니터링 도구이며, 파일 시스템이 변경될 때 커널에 의해 탐지할 수 있다. 하지만 dnotify는 디렉터리에 한정되어 모니터링이 가능하고, 모니터링 대상이 되는 각각의 디렉터리에 대하여 file descriptor를 사용해야 하기 때문에 다수의 file descriptor가 요구된다. 따라서 그 결과 병목현상이 발생하고, file descriptor로 인하여 디바이스는 마운트 해제가 불가능한 문제가 발생하였다. 또한 파일 시스템 변경 이벤트에 대하여 상태 정보를 제공하지 않아 디렉터리에 어떠한 변경이 일어났는지 확인하기 위해서는 함수를 사용하여 캐시와 비교하는 과정이 필수적이다.

2.1.2 inotify

inotify는 파일 시스템 변경 이벤트에 대한 리눅스 커널 서브 시스템이며, 모니터링 대상이 되는 파일 또는 디렉터리를 inotify_add_watch에 등록하면 해당 inode는 마킹되어 파일 시스템 변경을 위해 inode를 사용할 때마다 breaking 되어 모든 파일 시스템 객체를 모니터링 할 수 있도록 동작한다. inotify는 dnotify의 한계점을 개선하여 등장하였다. 파일과 디렉터리 모니터링이 가능하고, 하나의 file descriptor를 사용하며 inotify의 file descriptor는 시스템 호출을 통해 수행되어 마운트 해제 문제를 해결하였다. 또한 파일 시스템 변경 이벤트에 대하여 접근, 수정, 메타데이터 변경, 이동, 삭제 등의 상태 정보를 제공하며, 파일 시스템의 변경을 탐지하기 위하여 일정 주기마다 모니터링 할 필요가 없기 때문에 오버헤드가 거의 없는 것이 특징이다. 하지만 모니터링 대상이 되는 디렉터리의 하위 디렉터리에 대한 파일 시스템 변경 이벤트는 모니터링이 불가능하다는 한계점을 가지고 있다.

2.1.3 fanotify

fanotify는 파일 시스템 변경 이벤트에 대한 알림 및 차단 기능을 제공한다. inotify의 한계점이었던 모니터링 대상이 되는 디렉터리의 하위 디렉터리에 대한 파일 시스템 변경 이벤트 모니터링을 지원하며, 파일 시스템의 모든 객체를 모니터링 가능하도록 한다. fanotify에는 두 가지 비트 마스크가 존재하는데, Mark 마스크와 Ignore 마스크이다. 마크 마스크는 이벤트가 생성될 파일 활동을 정의하고, 무시 마스크는 이벤트가 생성되지 않는 활동을 정의한다. 또한 파일에 대한 접근 권한을 부여할 수 있다. 하지만 생성, 이동, 삭제 등과 같은 파일 시스템 변경 이벤트는 지원되지 않는 등 이벤트 상태 정보 제공이 제한되어있는 한계점을 가지고 있다.

2.2 클라우드 컴퓨팅 환경에서의 이벤트 모니터링 연구

2.2.1 Cloud-inotify

Wolfgang Richter가 제안한 IaaS 클라우드 컴퓨팅 환경에서의 파일 시스템 변경 모니터링 기술인 cloud-inotify는 publish-subscribe 네트워크 기반 메시징 시스템이다. cloud-inotify는 게스트 머신의 가상화된 디스크를 디스크 크롤러를 통해 파일 시스템 데이터 구조의 인덱스를 생성하고, 모니터링하려는 대상 디렉터리와 하위 디렉터리 및 포함된 파일에 대한 모든 업데이트를 네트워크를 통해 등록된 모니터링 응용 프로그램으로 메시지를 통해 복제되어 실행된다. 하지만 호스트 머신과 게스트 머신에 대하여 디스크에 데이터를 동기화해야 하는 작업이 필수적임에 따른 시스템 성능 오버헤드가 발생하고, 동시에 많은 파일 시스템 변경 이벤트가 일어나게 되면 쓰기 오버헤드가 최대 39.5%가 발생하는 것을 확인하였다. 또한 로그 파일을 모니터링하는 데 있어 평균 1sec의 지연 시간이 발생한다는 한계점을 가지고 있다.

해당 논문은 IaaS 클라우드 컴퓨팅 환경에서 구동되는 가상 머신에 대한 파일 시스템 변경 이벤트를 에이전트 없이 모니터링하는 목적을 가지고 있어, 본 논문에서 제안하는 시스템인 PaaS 클라우드 컴퓨팅 환경에서 구동되는 컨테이너에 대한 파일 시스템 변경 이벤트 모니터링 시스템과는 모니터링이 수행되는 기반 환경이 다르기 때문에 시스템 성능 측면에서 비교하기에 적합하지 않았다.

2.2.2 SAaaS(Security Audit as a Service)

Frank Doelitzscher은 클라우드 환경을 위한 에이전트 기반의 클라우드 인스턴트 탐지 시스템(SAaaS)을 제안하였다. 이는 클라우드 관련 보안 문제 및 서비스 사고 탐지를 위해 inotify를 활용하여 iNotify-agent를 구현함으로써 모니터링을 수행하였다. 이 논문에서는 inotify의 파일 시스템 이벤트 알림 기능을 사용하여 클라우드 사용자가 가상 머신에 로그인하여 일부 구성 파일을 변경하면 iNotify-agent가 이를 즉시 감지하도록 하였고, 감지된 이벤트를 로컬 Aggregator 에이전트로 전송하고, 허용된 이벤트인지 식별하여 가상 머신의 보안 상태를 확인하도록 설계하였다.

위 논문은 inotify를 활용하여 에이전트를 구현하였고, 에이전트를 가상 머신에 설치하여 클라우드의 보안 서비스를 제공하는 데 목적을 가지고 있기 때문에 본 논문에서 제안한 agent-free 모니터링 시스템과는 시스템 성능 측면에서의 비교를 수행하기에 적합하지 않았다.

3. 클라우드 환경에서의 컨테이너 친화적인 파일 시스템 이벤트 탐지 기술

본 장에서는 클라우드 컴퓨팅 환경의 호스트 머신 레벨에서 다수의 컨테이너에 대한 파일 시스템 변경 이벤트 탐지를 위한 요구사항을 도출하고, 도출된 요구사항을 바탕으로 시스템을 설계 및 구현하였다.

3.1 요구사항 도출

클라우드 컴퓨팅 환경에서 구동되는 수백, 수천 개의 컨테이너에 대한 파일 시스템 변경 이벤트 모니터링 시스템의 성능 효율적인 설계를 위해서는

다음과 같은 두 가지 요구사항이 충족되어야 한다.

첫째, 다수의 컨테이너에 대한 모니터링을 시스템 성능 효율적으로 수행하기 위해서는 생성, 구동, 정지, 종료와 같은 컨테이너 동작 상태와 같은 컨테이너의 라이프 사이클을 이해하는 것이 선행되어야 한다. 이후 컨테이너가 구동되지 않는 상태에서는 모니터링 역시 수행되지 않도록 설계하여 적재적소에 모니터링이 수행되도록 시스템 성능 효율을 높이는 것이 중요하다. 또한 일반적으로 수행되는 모니터링 구동 프로세스는 컨테이너가 구동된 이후 모니터링이 수행되기 때문에 컨테이너 구동 이후부터 모니터링이 수행되기 전까지 모니터링이 수행되지 않는 음영 영역이 발생한다. 이러한 음영 영역에서는 컨테이너에 대한 모니터링이 수행되지 않기 때문에 컨테이너가 구동된 시점부터 컨테이너가 정지되는 시점까지의 모든 순간이 모니터링이 되지 않아 정확한 모니터링을 수행했다고 볼 수 없다. 따라서 컨테이너의 동작 구조와 특성을 이해하고 모니터링 프로세스를 컨테이너에 최적화하여 정확도를 높일 수 있는 모니터링 알고리즘이 필요하다.

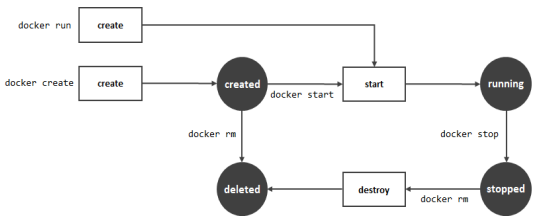
둘째, 컨테이너 구동 이후 사용자에 의해 발생하는 파일 시스템 변화에 대한 이벤트를 모니터링하기 위해서는 기존 파일 시스템 이벤트를 탐지하는데 주로 사용되었던 리눅스 커널의 서브 시스템인 inotify를 활용할 수 있다. 하지만 본 논문에서 inotify를 활용하기 위해서는 inotify가 가진 한계점인 모니터링 대상 디렉터리의 하위 디렉터리의 파일 시스템 변경 이벤트에 대하여 모니터링이 불가능하다 한계점을 해결해야 한다.

3.2 컨테이너 파일 시스템 변경 이벤트 탐지 시스템의 설계 및 구현

3.2.1 모니터링 성능 효율과 정확성을 높인 컨테이너를 위한 모니터링 알고리즘 설계

(1) 시스템 성능 효율을 높이기 위한 모니터링 알고리즘 설계

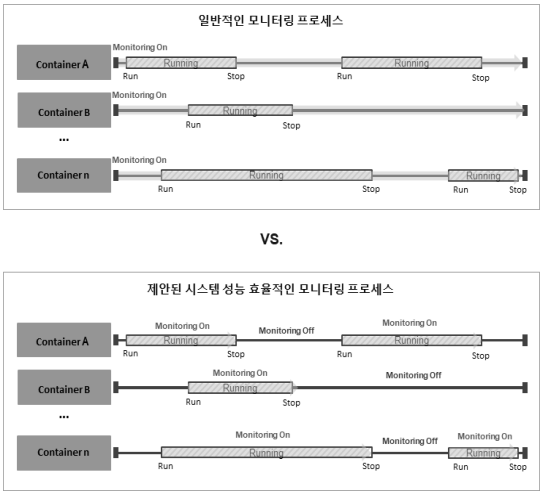
클라우드 컴퓨팅 환경에서는 수백, 수천 개의 컨테이너가 구동되며, 각각의 컨테이너들은 구동, 정지 및 종료 시점이 동일하지 않아 구동되는 기간 또한 다르다. 따라서 클라우드 컴퓨팅 환경에서 컨테이너를 효율적으로 모니터링하기 위해서는 Docker 컨테이너의 라이프 사이클 구조에 대하여 파악해야 한다.



(그림 1) 컨테이너 라이프 사이클

Docker 컨테이너의 라이프 사이클은 (그림 1)과 같이 크게 생성, 구동, 정지 및 종료로 나눌 수 있다. 먼저 Docker 컨테이너를 생성하는 방법에는 두 가지가 있는데, 첫 번째 방법은 docker create 명령어를 사용하여 컨테이너를 생성하는 것으로, 추후 컨테이너를 구동하기 위해서는 docker start 명령어를 통해 구동해줘야 한다. 두 번째 방법은 docker run 명령어를 사용하여 컨테이너 생성과 함께 구동되도록 한다. 컨테이너 구동이 시작된 이후에는 컨테이너 정지가 가능하며 이는 컨테이너 종료와는 별개로 추후 다시 구동할 수 있다. 또한 컨테이너를 종료하기 위해서는 반드시 구동 중인 컨테이너를 정지 후 종료하여야 한다. 이와 같은 라이프 사이클을 가진 클라우드 컴퓨팅 환경에서 구동되는 수많은 컨테이너들은 언제 구동 또는 정지될지 모르기 때문에 컨테이너가 구동되는 시작 시점과 정지 시점을 정확히 인지하는 것은 모니터링 성능 효율에 있어 굉장히 중요한 역할을 담당한다.

(그림 2)는 컨테이너 라이프 사이클에 대한 이해를 돕기 위하여 컨테이너 모니터링 프로세스를 일반적으로 구동되는 모니터링 알고리즘에 적용했을 때와 본 논문에서 제안한 시스템 성능 효율적인 모니터링 알고리즘에 적용했을 때의 프로세스를 비교하여 그림으로 나타낸 것이다.

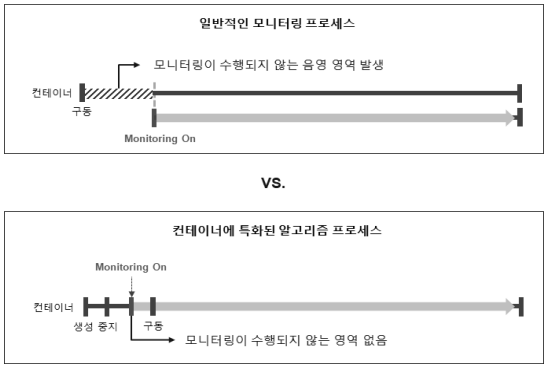


(그림 2) 일반적인 모니터링 프로세스와 제안된 시스템 성능 효율적인 모니터링 프로세스 비교

(2) 모니터링 정확성을 높이기 위한 알고리즘 설계

일반적인 모니터링 프로세스는 모니터링 대상 컨테이너가 구동된 이후 모니터링을 실행하여 이벤트를 탐지하는 순서로 구동된다. 하지만 이러한 일반적인 모니터링 프로세스는 모니터링 대상이 되는 컨테이너가 구동된 이후 모니터링이 실행되기 때문에 컨테이너가 구동된 이후부터 모니터링이 실행되기 전까지 모니터링이 수행되지 않는 음영 영역이 반드시 발생한다는 한계점을 가지고 있다. 이러한 이유로 일반적인 모니터링 프로세스는 컨테이너 구동 시점부터 종료 시점까지의 모든 것을 모니터링 한다고 보기 어려우며, 이는 모니터링 정확성이 떨어지는 결과를 가져오게 된다. 따라서 모니터링의 정확성을 높이기 위하여 컨테이너 생성 후 컨테이너를 잠시 정지시키고, 모니터링 실행 후 컨테이너를 다시 구동하도록 하여 모니터링이 수행되지 않는 음영 영역이 발생하지 않는 컨테이너에 특화된 무손실 모니터링이 가능하도록 하였다.

(그림 3)는 컨테이너 모니터링 프로세스를 일반적인 모니터링 알고리즘에 적용하였을 때와 제안된 컨테이너에 특화된 무손실 모니터링 알고리즘에 적용하였을 때의 모니터링이 수행되지 않는 음영 영역을 비교하여 그림으로 나타낸 것이다.



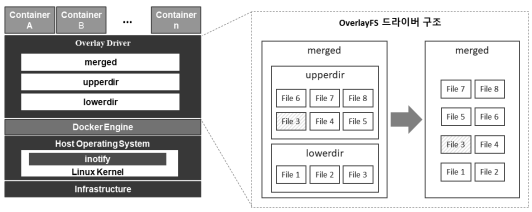
(그림 3) 알고리즘에 따른 모니터링이 수행되지 않는 음영 영역 비교

3.2.2 inotify를 활용한 모니터링 메커니즘 설계

기존의 파일 변화를 감지하는 데 주로 사용되었던 inotify는 모니터링 대상 디렉터리의 하위 디렉터리에 대한 파일 시스템 변경 이벤트는 모니터링 하지 못한다는 한계점을 가지고 있다. 이러한 한계점은 컨테이너에 특화된 무손실 모니터링 알고리즘을 통해 모니터링 시스템 성능 효율을 높이고, 정확성을 높였다고 하더라도 이 한계점을 해결하지 못한다면 제대로 된 모니터링을 수행한다고 할 수 없다. 따라서 이러한 inotify의 한계점을 해결하기 위하여 Docker 컨테이너의 OverlayFS 드라이버 구조 분석을 바탕으로 시스템 성능 효율적인 하위 디렉터리의 파일 시스템 변경 이벤트 모니터링이 가능하도록 하는 메커니즘을 제안한다.

Docker 컨테이너의 OverlayFS은 리눅스를 위한 UnionFS(Union File System) [22, 23]으로, 여러 개의 레이어를 하나의 파일 시스템으로 사용할 수 있도록 제공해준다. OverlayFS는 기본적으로 (그림 4)와 같이 Docker 컨테이너에 lowerdir, upperdir, merged를 계층화하여 레이어로 구성된다. lowerdir은 Docker의 read-only 이미지 레이어이다. upperdir은 컨테이너 레이어로, 컨테이너에서 변경되는 모든 스토리지 정보가 해당 레이어에 저장된다. merged는 lowerdir과 upperdir 데이터가 통합되어 존재하는 컨테이너 마운트 레이어이다. 이와 같은 OverlayFS의 레이어 구조에 따라 본

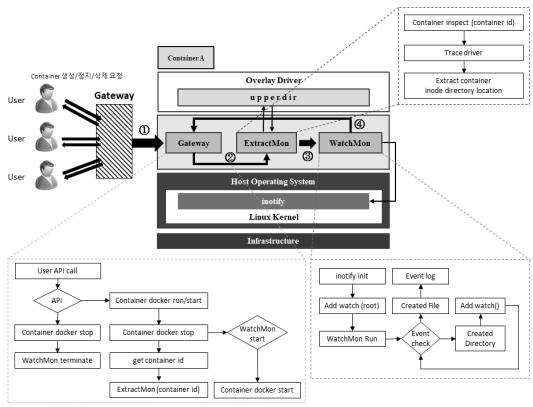
논문에서는 OverlayFS 드라이버의 구조 분석을 통하여 모든 레이어를 모니터링 할 필요 없이 upperdir 레이어만을 모니터링하여 컨테이너 사용자가 변경하는 파일 시스템 변경 이벤트만을 시스템 성능 효율적으로 모니터링 할 수 있도록 한다.



(그림 4) OverlayFS 드라이버 구조

3.2.3 컨테이너 파일 시스템 변경 이벤트 탐지 시스템 프레임워크

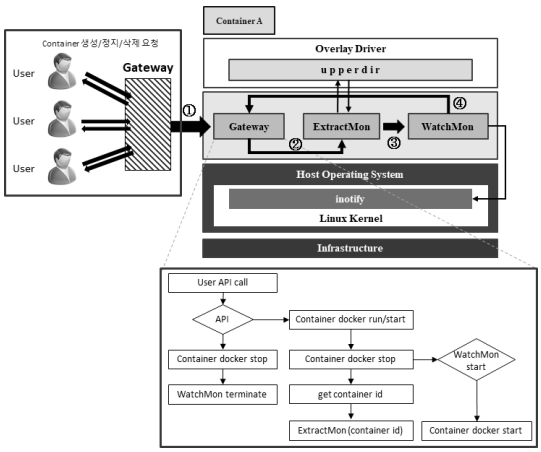
본 논문에서 제안한 컨테이너 파일 시스템 변경 이벤트 모니터링 시스템을 PaaS 클라우드 컴퓨팅 환경에 적용하기 위하여 (그림 5)과 같은 프레임워크를 구축하였다. 제안된 시스템은 기능별로 모듈화하여 설계하였으며, Gateway 모듈, ExtractMon 모듈, WatchMon 모듈로 총 3가지 모듈을 제공한다. 다음은 각 모듈에 대한 설명이다.



(그림 5) 제안된 파일 시스템 변경 이벤트 탐지 시스템 구성도

(1) Gateway 모듈

전체 프레임워크에서 Gateway 모듈의 구성과 역할은 (그림 6)와 같이 나타낼 수 있다. 웹에서 사용자의 컨테이너 구동에 대한 모든 API Call을 전달받고, 이에 따라 컨테이너의 라이프 사이클을 어떠한 상태로 변경시키는지 확인한다. 사용자 인터페이스 요청에 따라 컨테이너에 대한 생성 요청이 들어오면 Docker Client를 통해 컨테이너를 생성하고, 생성한 컨테이너에 대한 id를 추출한다. 또한 컨테이너 정지 또는 종료 명령이 전달되면 구동 중인 WatchMon을 종료시킨다. 이와 같이 Gateway 모듈은 웹에서 컨테이너에 대한 모든 사용자 인터페이스를 Gateway로 받아 컨테이너를 동작하도록 구현하였기 때문에 컨테이너 구동, 정지 및 종료 시점을 정확히 파악할 수 있도록 함으로써 시스템 성능 효율이 높은 모니터링이 가능하도록 구현하였다. 만약 컨테이너가 오류를 일으켜 정지 상태가 되었다고 하더라도 Gateway를 통해 정상적으로 입력된 명령이 아니기 때문에 모니터링 구동에 영향을 주지 않는다.

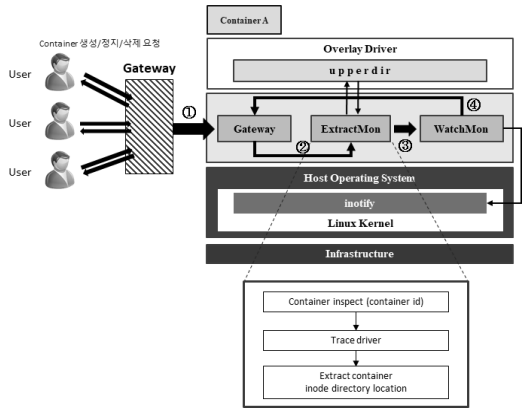


(그림 6) Gateway 모듈 구성도

(2) ExtractMon 모듈

ExtractMon 모듈은 (그림 7)과 같이 나타낼 수 있다. 컨테이너의 상세 정보를 확인하기 위하여 inspect 명령어를 통해 upperdir 레이어의 위치를

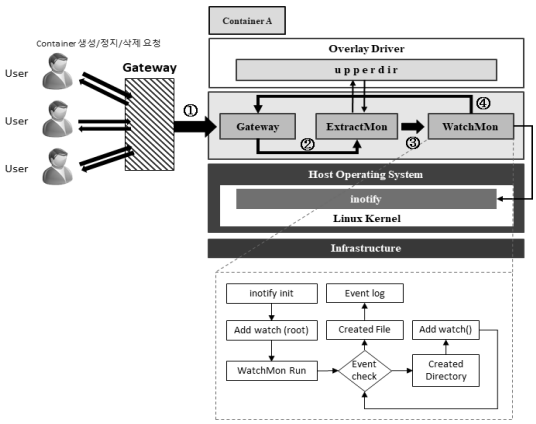
확인하고, 해당 정보 값을 Gateway 모듈에 리턴한다. upperdir 레이어는 컨테이너 사용자가 변경한 파일 시스템만을 저장하기 때문에 upperdir 레이어의 위치 정보 값을 통해 사용자에 의한 파일 시스템 변경 이벤트만 탐지할 수 있도록 하여 모니터링 시스템 성능 효율을 높였다.



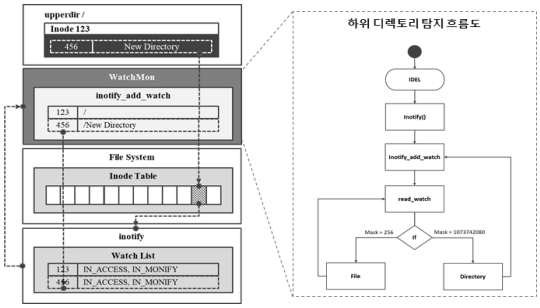
(그림 7) Extract 모듈 구성도

(3) WatchMon 모듈

WatchMon 모듈은 (그림 8)과 같으며, 하위 디렉터리의 파일 시스템 변경 이벤트를 모니터링하는 흐름도는 (그림 9)와 같다. Gateway 모듈에서 upperdir 레이어 위치를 받아와 upperdir의 루트 디렉터리에 대하여 inotify 모니터링을 수행한다. 루트 디렉터리의 하위 디렉터리를 모니터링하기 위해서는 루트 디렉터리의 파일 시스템 변경 이벤트가 탐지되었을 때 inotify의 모니터링 로그 정보에서 변경된 파일에 대한 이벤트 정보와 마스크 값을 확인하여 파일과 디렉터리를 구분한다. 이때 활용되는 마스크 값은 리눅스에서 지정해 놓은 마스크 값이다. 만약 모니터링 로그 정보 값이 디렉터리 생성이라면 하위 디렉터리가 생성된 것이기 때문에 이를 모니터링하기 위하여 inotify_add_watch에 생성된 하위 디렉터리 위치를 추가하고 inotify를 실행한다.



(그림 8) WatchMon 모듈 구성도



(그림 9) 하위 디렉터리 모니터링 흐름도

4. 성능 평가

본 장에서는 논문에서 제안한 컨테이너에 특화된 파일 시스템 변경 이벤트 탐지 시스템을 통해 PaaS 클라우드 컴퓨팅 환경에서 구동되는 컨테이너 파일 시스템 이벤트를 탐지하는 데 있어 발생하는 시스템 오버헤드와 본 논문에서 제안한 컨테이너에 특화된 무손실 모니터링 알고리즘의 정확성에 대하여 성능 평가한다.

4.1 실험환경

본 논문에서 제안한 프레임워크의 성능을 평가하기 위하여 오픈소스 컨테이너 플랫폼인 Docker 환경을 구축하여 실험을 진행하였으며, 실험 환경은 <표 1>과 같다. 실험에서 사용된 호스트 서버의 운영체제는 Ubuntu 17.10 x64이고, CPU는 Intel(R)

Xeon(R) CPU E5-2690 v4 @ 2.60GHz을 사용하였다. 메모리는 256GB를 사용하였고, 디스크는 하드디스크 ST4000NM0165 1TB를 사용하였다. 실험에서 사용된 컨테이너의 Base 이미지는 Ubuntu 18.04를 사용하였고, Docker Version은 Docker version 18.06.1-ce를 사용하여 실험을 진행하였다.

〈표 1〉 실험환경

호스트	
운영체제	Ubuntu 17.10 x64
CPU	Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz
메모리	256GB
디스크	ST4000NM0165 1TB
컨테이너	
컨테이너 Base 이미지	Ubuntu 18.04
Docker Version	Docker version 18.06.1-ce

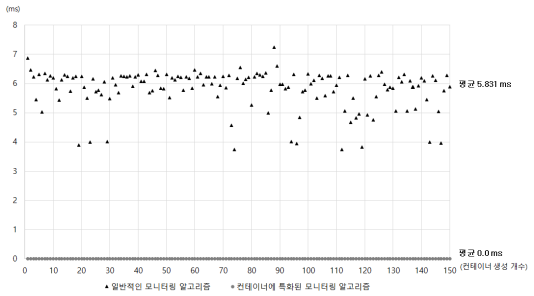
4.2 실험내용

4.2.1 알고리즘에 따른 컨테이너 구동 시간 및 음영 영역 측정

본 논문에서는 모니터링이 수행되지 않는 음영 영역을 최소화하기 위한 방안 중 하나로써 컨테이너 환경에 특화된 무손실 모니터링 알고리즘을 고안하여 알고리즘에 따른 구동 소요 시간을 측정하고, 그 성능을 평가하기 위해 두 가지 알고리즘으로 실험을 진행하였다.

(1) 컨테이너 생성부터 모니터링 구동까지 소요되는 시간 측정

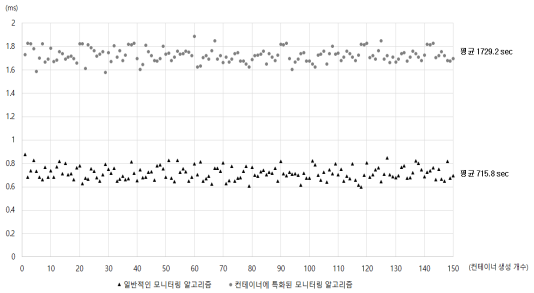
실험 결과는 (그림 10)과 같다. x축은 실험 횟수를 의미하며, y축은 컨테이너 생성 요청부터 모니터링 구동까지 소요되는 시간, 즉 모니터링이 수행되지 않는 음영 영역에 대한 소요 시간을 ms(millisecond)로 나타내었다. 총 150번의 실험을 진행하였으며, 컨테이너 생성부터 모니터링 구동까지 일반적인 모니터링 알고리즘은 평균 5.831ms가 소요되었고, 컨테이너에 특화된 무손실 모니터링 알고리즘은 평균 0.0ms가 소요되었음을 확인할 수 있다.



(그림 10) 컨테이너 생성부터 모니터링 구동까지 소요되는 시간 측정 결과

(2) 컨테이너 생성 요청부터 컨테이너 구동까지 소요되는 시간 측정

실험 결과는 (그림 11)과 같다. x축은 실험 횟수를 의미하며, y축은 컨테이너 생성 요청부터 컨테이너 구동까지 소요되는 시간을 ms로 나타내었다. 총 150번의 실험을 진행하였으며, 사용자로부터의 컨테이너 생성 요청부터 컨테이너 구동까지 일반적인 모니터링 알고리즘은 평균 715.8ms가 소요되었고, 컨테이너에 특화된 무손실 모니터링 알고리즘은 평균 1729.2ms가 소요됨을 확인할 수 있다.



(그림 11) 컨테이너 생성 요청부터 컨테이너 생성까지 소요되는 시간 측정 결과

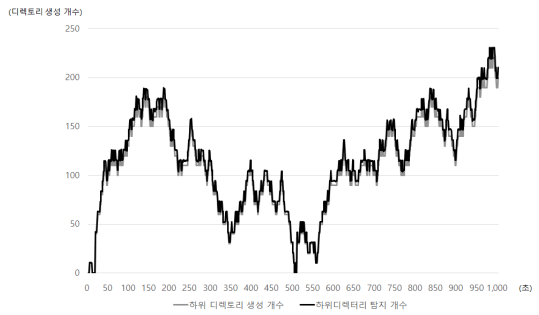
4.2.2 하위 디렉터리 모니터링 성능 측정 실험

본 실험에서는 inotify의 한계점을 해결하기 위하여 구현한 하위 디렉토리 파일 시스템 변경 이벤트 모니터링 메커니즘에 대한 성능 평가를 수행하였다. 실험을 진행하기에 앞서, 하나의 무작위 디렉터리를 생성하기 위한 워크로드를 생성하였다. 이

워크로드를 활용하여 무작위로 디렉터리를 생성하고 하위 디렉터리의 파일 시스템 변경 이벤트 모니터링 메커니즘을 수행하였을 때 트레이스(추적) 그래프가 동일하게 나타난다면 하위 디렉터리 탐지가 정확히 수행되고 있다고 예상할 수 있다.

(1) 하위 디렉터리 모니터링 커버리지 검증

실험 결과는 (그림 12)와 같다. 0sec부터 1,000sec까지 하위 디렉터리를 생성 및 삭제에 따른 실험을 진행하였다. (그림 12)에서 x축이 시간이고, y축이 생성된 하위 디렉터리 개수라고 하였을 때, 하위 디렉터리 개수 변화와 시간 흐름에 따라 디렉터리 탐지까지 평균 0.4ms의 지연 시간이 발생하였지만, 하위 디렉터리를 모두 탐지하는 것을 확인할 수 있었다.



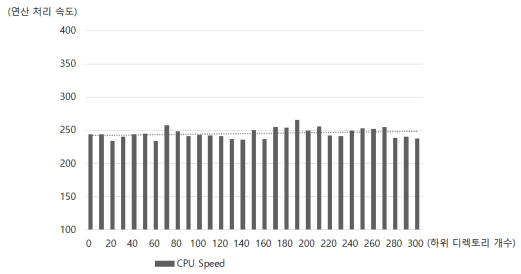
(그림 12) 하위 디렉터리 모니터링 커버리지 검증 결과

4.2.3 하위 디렉터리 개수에 따른 모니터링 시스템 오버헤드 측정

본 실험에서는 하위 디렉터리 모니터링에 따른 컨테이너에 대한 시스템 오버헤드를 측정하기 위하여 컨테이너에서 0개부터 300개까지의 하위 디렉터리 생성 후 각각의 디렉터리를 모니터링하였을 때의 하위 디렉터리 모니터링에 따른 컨테이너의 시스템 성능 오버헤드를 평가하였다. 측정 방법은 리눅스 서버 벤치마크 프로그램인 sysbench [24]를 사용하였으며 CPU, 메모리 읽기 및 쓰기, 디스크 읽기 및 쓰기에 대한 시스템 오버헤드 성능을 측정하였다.

(1) CPU 오버헤드 측정

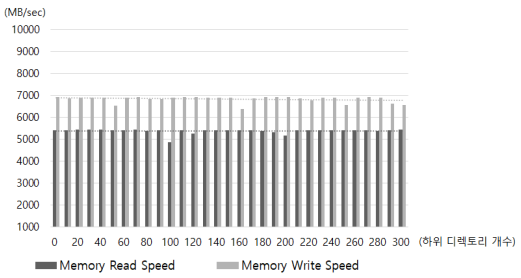
CPU 오버헤드 측정 실험 결과는 (그림 13)과 같다. 하위 디렉터리 개수가 0일 때, CPU 초당 이벤트 연산 처리 속도는 243.82가 측정되었다. 이를 기준으로 하위 디렉터리 300개로 늘려 측정하였을 때를 살펴보면 CPU 초당 이벤트 연산 처리 속도는 237.49가 측정되었다. 아래 그림과 같이 디렉터리가 증가함에 따라 약간의 노이즈에 의한 처리량 증가 및 감소는 보이지만, 0.55% 미만의 오버헤드만이 발생된다는 것이 실험을 통해 도출되었다.



(그림 13) CPU 오버헤드 측정 결과

(2) 메모리 읽기 및 쓰기 오버헤드 측정

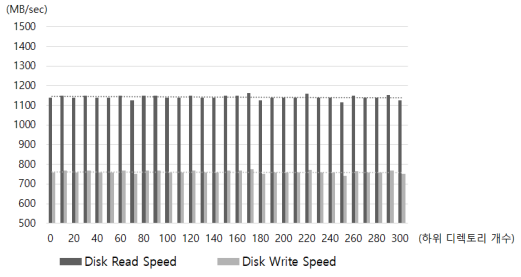
메모리 읽기 및 쓰기 오버헤드 측정 실험 결과는 (그림 14)와 같다. 하위 디렉터리 개수가 0일 때, 메모리 읽기 속도는 5433.81MB/sec이고, 쓰기 속도는 6928.52MB/sec로 측정되었다. 이를 기준으로 하위 디렉터리 300개로 늘려 측정하였을 때를 살펴보면 메모리 읽기 속도는 평균 5433.93MB/sec이고, 쓰기 속도는 평균 6928.61MB/sec를 확인할 수 있다. 따라서 아래 그림과 같이 디렉터리가 증가함에 따라 약간의 노이즈에 의한 메모리 쓰기 및 읽기 속도의 증가 및 감소는 보이지만, 메모리 읽기 속도는 평균 0.002% 오버헤드를 보였고, 메모리 쓰기 속도는 평균 0.001%의 오버헤드가 발생함을 확인할 수 있었다.



(그림 14) 메모리 읽기 및 쓰기 오버헤드 측정 결과

(3) 디스크 읽기 및 쓰기 오버헤드 측정

디스크 읽기 및 쓰기 오버헤드 측정 실험 결과는 (그림 15)와 같다. 하위 디렉터리 개수가 0일 때, 디스크 읽기 속도는 1138.86MB/sec이고, 쓰기 속도는 759.24MB/sec로 측정되었다. 이를 기준으로 하위 디렉터리 300개로 늘려 측정하였을 때를 살펴보면 디스크 읽기 속도는 평균 1142.57MB/sec이고, 쓰기 속도는 평균 761.60MB/sec가 측정되었다. 따라서 아래 그림과 같이 디렉터리가 증가함에 따라 약간의 노이즈에 의한 디스크의 쓰기 및 읽기 속도의 증가 및 감소는 보였지만, 디스크 읽기 속도는 평균 0.32%의 오버헤드와 디스크 쓰기 속도는 평균 0.31%의 오버헤드를 보였다.



(그림 15) 디스크 읽기 및 쓰기 오버헤드 측정 결과

5. 결론

본 논문에서는 다수의 컨테이너가 구동되고 있는 클라우드 컴퓨팅 환경에서 파일 시스템 변경 이벤트를 탐지하기 위한 다음과 같은 두 가지 요구사항을 도출하고 이를 바탕으로 설계 및 구현하였다.

첫째, 클라우드 컴퓨팅 환경에서 구동되는 수백, 수천 개의 컨테이너를 모니터링하는 것은 시스템 성능에 있어서 상당한 오버헤드가 발생한다. 또한 일반적인 모니터링 알고리즘 프로세스는 컨테이너 구동 이후 모니터링이 수행되기 전까지, 모니터링이 수행되지 않는 음영 영역이 발생하는 한계점을 가지고 있다. 따라서 이와 같은 문제점을 해결하기 위해 컨테이너의 라이프 사이클 이해를 바탕으로 컨테이너의 구동, 정지 및 종료 시점을 인지함으로써 컨테이너가 구동되는 중에만 모니터링이 수행될 수 있도록 하여 모니터링의 시스템 성능 효율을 높였다. 또한 컨테이너 모니터링에 특화된 무손실 모니터링 알고리즘을 고안하여 모니터링이 수행되지 않는 음영 영역을 완전히 삭제시킴으로써 모니터링의 정확성을 높이도록 하였다. 이는 실험을 통하여 모니터링 음영 영역 삭제를 증명하였고 이로써 모니터링 정확성 향상을 확인하였다.

둘째, 기존의 파일 시스템 변화를 탐지하는 데 주로 사용되었던 inotify는 모니터링하는 대상 디렉터리의 하위 디렉터리에 대한 파일 시스템 변경 이벤트를 탐지하지 못한다는 한계점을 가지고 있다. 따라서 본 논문에서는 이러한 inotify의 문제점을 컨테이너의 OverlayFS 드라이버 구조 분석을 통하여 시스템 성능 효율적으로 하위 디렉터리 파일 시스템 이벤트 모니터링이 가능하도록 하는 메커니즘을 제안하였다. 이는 실험을 통하여 하위 디렉터리 모니터링에 대한 정확성과 컨테이너 시스템 오버헤드를 평가하였다. 먼저, 하위 디렉터리 모니터링의 정확성은 무작위 디렉터리 생성 워크로드를 만들어 실험을 진행하였고, 실험 결과 하위 디렉터리를 탐지하는 데 있어 평균 0.4ms의 지연 시간은 발생하였지만 생성된 하위 디렉터리에 대하여 모두 탐지하였음을 확인할 수 있었다. 하위 디렉터리 모니터링 개수 증가에 따른 컨테이너의 시스템 오버헤드 성능 평가에서는 디렉터리가 증가함에 따라 약간의 노이즈에 의한 처리량 증가 및 감소는 보였지만, 컨테이너의 CPU, 메모리 읽기 및 쓰기, 디스크 읽기 및 쓰기 속도에 아주 미세한 영향만을 발생시킨다

는 것이 실험을 통해 도출되었다.

이를 통해 본 논문에서 제안한 시스템은 향후 다수의 컨테이너가 구동되는 클라우드 컴퓨팅 환경에서 파일 시스템 변경 이벤트 탐지에 대하여 효과적으로 활용이 가능할 것으로 기대할 수 있다.

향후 연구에서는 현재의 연구 결과를 확장하여 클라우드 컴퓨팅 환경에서의 컨테이너 이벤트 탐지에 따른 Long-Term 타임라인 생성 시스템을 연구할 계획이다.

▌참고문헌

〔1〕 Varia, Jinesh, and Sajee Mathew. "Overview of amazon web services." Amazon Web Services (2014).

〔2〕 Kim Young-won, "AWS server outage in Seoul leads to online glitches", http://www.theinvestor.co.kr/view.php?ud=20181122000711

〔3〕 YouTube, L. L. C. "YouTube." Retrieved 27 (2011): 2011.

〔4〕 PETER MARTINEZ, "YouTube fixes 'access issues' that caused widespread outage", https://www.cbsnews.com/news/youtube-outage-access-issues-today-2018-10-16-live-updates/

〔5〕 Cinque, Marcello, Domenico Cotroneo, and Antonio Pecchia. "Event logs for the analysis of software failures: A rule-based approach." IEEE Transactions on Software Engineering 39.6 (2013): 806-821.

〔6〕 Suneetha, K. R., and Raghuraman Krishnamoorthi. "Identifying user behavior by analyzing web server access log file." IJCSNS International Journal of Computer Science and Network Security 9.4 (2009): 327-332.

〔7〕 Bass, Len, Ingo Weber, and Liming Zhu. DevOps: A software architect's perspective. Addison-Wesley Professional, 2015.

〔8〕 CALLANAN, M., AND SPILLANE, A. Devops: Making it easy to do the right thing. IEEE Software 33, 3 (May 2016), 53 - 59.

〔9〕 Qian, Ling, et al. "Cloud computing: An overview." IEEE International Conference on Cloud Computing. Springer, Berlin, Heidelberg, 2009.

〔10〕 Merkel, Dirk. "Docker: lightweight linux containers for consistent development and deployment." Linux Journal 2014.239 (2014): 2.

〔11〕 Bernstein, David. "Containers and cloud: From lxc to docker to kubernetes." IEEE Cloud Computing 3 (2014): 81-84.

〔12〕 Felter, Wes, et al. "An updated performance comparison of virtual machines and linux containers." Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On. IEEE, 2015.

〔13〕 Love, Robert. "Kernel korner: Intro to inotify." Linux Journal 2005.139 (2005): 8.

〔14〕 Love, Robert. Linux System Programming: Talking Directly to the Kernel and C Library. " O'Reilly Media, Inc.", 2013.

〔15〕 Jonathan Corbet, "The fanotify API", https://lwn.net/Articles/339399/

〔16〕 Richter, Wolfgang, et al. "Agentless cloud-wide streaming of guest file system updates." 2014 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2014.

〔17〕 Doelitzscher, Frank, et al. "An agent based business aware incident detection system for cloud environments." Journal of Cloud Computing: Advances, Systems and Applications 1.1 (2012): 9.

〔18〕 Merkel, Dirk. "Docker: lightweight linux con-

tainers for consistent development and deployment." Linux Journal 2014.239 (2014): 2.
〔19〕 Mizusawa, Naoki, et al. "Performance Improvement of File Operations on OverlayFS for Containers." 2018 IEEE International Conference on Smart Computing (SMARTCOMP). IEEE, 2018.
〔20〕 Hermans, Siem, and Patrick de Niet. "Docker overlay networks: Performance analysis in highlatency environments."
〔21〕 Zismer, Arne. "Performance of Docker Overlay Networks." University of Amsterdam (2016).
〔22〕 Quigley, David, et al. "Unionfs: User-and community-oriented development of a unification filesystem." Proceedings of the 2006 Linux Symposium. Vol. 2. 2006.
〔23〕 Wright, Charles P., and Erez Zadok. "Kernel korner: unionfs: bringing filesystems together." Linux Journal 2004.128 (2004): 8.
〔24〕 Kopytov, Alexey. "SysBench: a system performance benchmark." http://sysbench. sourceforge. net/ (2004).

저자소개

◆ **전우진**

- 2015년 대전대학교 정보보안학과 학사
- 2019년 세종대학교 정보보호학과 석사
- 관심분야: 클라우드 컴퓨팅, 시스템 보안 등

◆ **박기웅**

- 2005년 연세대학교 Computer Science 학사
- 2007년 KAIST Electrical Engineering 석사
- 2012년 KAIST Electrical Engineering 박사
- 2008년 Microsoft Research Asia, Wireless and Networking Grourp, Research Intern
- 2009년 Microsoft Research, Network Research Group, Graduate Research Fellow
- 2012년 국가보안기술연구소 연구원
- 2012년~2016년 대전대학교 정보보안학과 교수
- 2016년~현재 세종대학교 정보보호학과 교수
- 관심분야: 시스템 보안, 모바일-클라우드 컴퓨팅, 보안 프로토콜, 디지털 포렌식 등