학 위 논 문

cAdvisor를 이용한 Docker 컨테이너 모니터링 시스템 구현

Implementaition of Docker Container Monitoring System Using cAdvisor

호남대학교
컴퓨터공학과
이름 이 우 진
2025년 02월

학 위 논 문

cAdvisor를 이용한 Docker 컨테이너 모니터링 시스템 구현

Implementaition of Docker Container Monitoring System Using cAdvisor

호남대학교
컴퓨터공학과
이름 이 우 진
2025년 02월

cAdvisor를 이용한 Docker 컨테이너 모니터링 시스템 구현

Implementation of Docker Container Monitoring System Using cAdvisor

지도교수 오 명 훈

이 논문을 학위 청구논문으로 제출함

2024년 12월

호남대학교
컴퓨터공학과
이름 이 우 진

이 우 진의 학위 논문을 인준함

심사위원장　　최 광 미　　(서명 또는 날인)

심 사 위 원　　오 명 훈　　(서명 또는 날인)

심 사 위 원　　박 현 아　　(서명 또는 날인)

호남대학교
컴퓨터공학전공
이 우 진
2024년 12월

목차

[표 목차]

< 그림 목차 >

논문요약

cAdvisor를 이용한 도커 컨테이너 모니터링 시스템 구현

현대에는 빅데이터, 인공지능, 사물인터넷 등의 기술 발전으로 애플리케이션의 복잡성이 증가하고 개발 속도의 가속화가 요구되고 있다. 이에 따라 컨테이너 환경이 반복적인 작업을 효율적으로 처리할 수 있는 대안으로 주목받고 있다. 컨테이너는 개발과 배포 속도를 높이고, 환경 간의 불일치를 줄이며, 자원 소모를 줄여 다양한 인프라에서 효율적으로 활용할 수 있는 장점이 있다. 이러한 이유로 컨테이너 기술은 의료, 금융, 소매 등 다양한 산업에 적용되고 있으며, 그 시장 규모는 지속적으로 성장하고 있다.

컨테이너 환경의 중요성이 증가하면서, 성능 문제나 보안 사고를 사전에 식별하고 해결하기 위한 컨테이너 모니터링은 필수가 되었다. 모니터링을 통해 전체적인 상태를 실시간으로 확인하고, 성능 문제의 원인을 파악하며, 데이터 시각화 및 자동 경고 기능으로 컨테이너를 효율적으로 관리할 수 있다.

본 논문에서는 cAdvisor, Prometheus, Grafana를 활용하여 Docker 컨테이너 모니터링 시스템을 구현하였다. 각각의 도구는 다음과 같은 역할을 한다. cAdvisor는 Docker 컨테이너의 CPU, 메모리, 네트워크 사용량 등의 다양한 메트릭을 수집한다. Prometheus는 이러한 메트릭을 주기적으로 스크랩하여 시계열 데이터베이스에 저장하고, 이를 통해 모니터링 및 경고 시스템을 제공한다. Grafana는 Prometheus에 저장된 데이터를 가져와 시각화하여 사용자가 컨테이너의 상태를 직관적으로 파악할 수 있도록 한다. Docker 컨테이너 모니터링 시스템을 구축하는 과정에서 Docker Compose를 활용해 세 가지 도구를 각각의 컨테이너로 설정하고, 데이터 수집, 저장, 시각화가 자동으로 연동될 수 있도록 구성하였다. 또한 다양한 실험을 통해 이 모니터링 시스템이 각 컨테이너의 자원 사용량을 실시간으로 보여주는 것을 검증하였다. 예를 들어, CPU 부하를 가중시키는 task0, 메모리 사용량을 일시적으로 증가시키는 task1, 네트워크 트래픽을 유발하는 task2라는 세 개의 컨테이너를 동시에 실행하여, 각각의 자원 사용량 변화가 Grafana에서 어떻게 시각화되는지 확인하였다. 이를 통해 컨테이너 기반 애플리케이션의 성능과 안정성을 모니터링하고, 문제 발생 시 빠르게 대응할 수 있는 환경을 제공할 수 있음을 보여준다.

주제어 : Docker 컨테이너, 모니터링, cAdvisor, Prometheus, Grafana

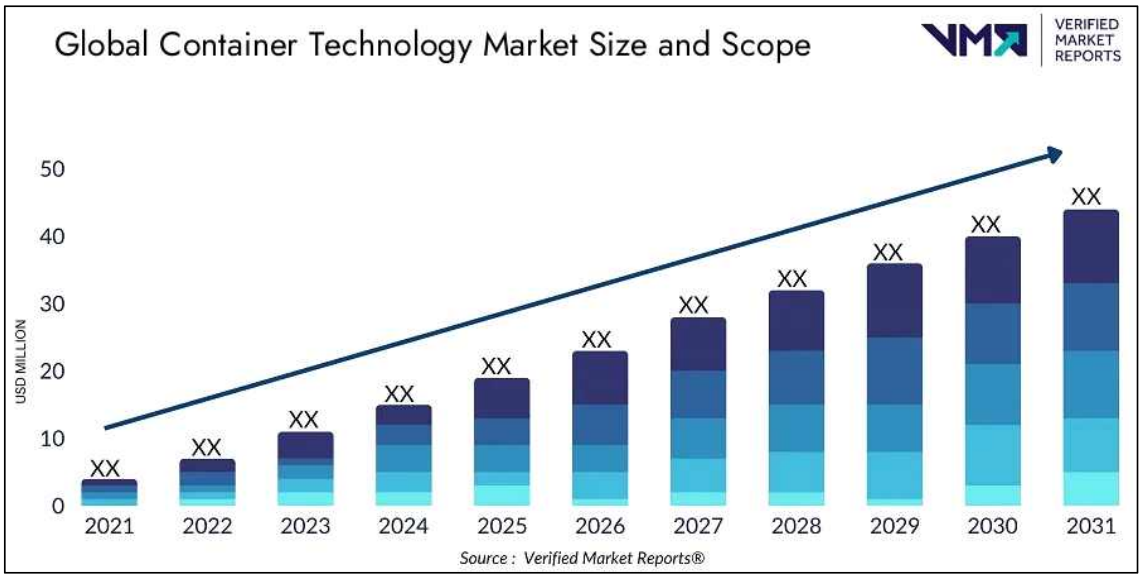
제1장 서 론

1.1 연구의 배경 및 목적

오늘날 빅데이터, 인공지능, 가상현실, 사물인터넷, 클라우드 등의 기술이 빠르게 발전함에 따라 애플리케이션의 복잡성이 증가하고 있으며, 애플리케이션 개발 속도 가속화에 대한 요구 또한 커지고 있다. 이에 따라 인프라, IT 팀, 그리고 개발 프로세스에 대한 부담도 늘어나고 있다. 이러한 문제를 해결하고 반복적인 작업을 효율적으로 처리하기 위한 대안으로 컨테이너 환경이 주목받고 있다. 컨테이너는 소프트웨어 개발 및 배포 과정에서 중요한 역할을 하고 있으며, 이를 통해 여러 문제를 완화하고 작업을 더 빠르게 처리할 수 있다. [1]

컨테이너는 소프트웨어 개발 및 배포 과정에 있어서 많은 장점이 있다. 첫째, 컨테이너를 활용하면 애플리케이션을 더 빠르게 개발하고 테스트할 수 있으며, 개발 환경과 운영 환경 간의 불일치를 줄임으로써 코드 변경과 배포 주기를 단축할 수 있다. 둘째, 경량화된 가상화 기술로 로컬 환경, 클라우드, 온프레미스 데이터센터 등 다양한 인프라 환경에서 손쉽게 배포 및 실행할 수 있다. 셋째, 컨테이너는 가상 머신보다 자원 소모가 적어 동일한 하드웨어에서 더 많은 애플리케이션을 실행할 수 있다. 넷째, 컨테이너는 인프라 관리와 배포 작업을 자동화함으로써 개발 주기를 가속화할 수 있다, 이와 같은 이유로 컨테이너 환경은 소프트웨어 개발 및 배포의 필수적인 요소로 자리매김하고 있으며, 그 중요성은 지속적으로 증가하고 있다. [2]

위와 같은 컨테이너 기술의 장점을 바탕으로 컨테이너 기술은 의료, 금융, 소매 등 다양한 산업에서도 적용된다. 예를 들어, 의료 분야에서는 컨테이너 기술을 활용하여 의료 데이터 관리를 효율적으로 할 수 있고, 금융 부문은 안전한 거래와 신속한 애플리케이션 업데이트를 위해 컨테이너 환경을 사용할 수 있으며, 소매 기업은 컨테이너 기술을 이용하여 재고 관리 시스템을 고객들에게 빠르게 배포할 수 있다. 이처럼 컨테이너 기술이 다양한 산업에 적용됨에 따라 아래 [그림 1-1]과 같이 컨테이너 기술 시장의 규모는 현재까지 성장해 왔으며 앞으로도 꾸준히 성장할 것으로 보고 있다. [3]



[그림 1-1] 글로벌 컨테이너 기술 시장 규모 증가 예상치

(출처: https://www.verifiedmarketreports.com/ko/product/container-technology-market/)

컨테이너 환경의 중요성과 컨테이너 기술의 사용이 늘어남에 따라 성능 문제, 소프트웨어 버그, 보안 사고 등을 사전에 식별하고 해결하는 것이 중요해졌다. 이에 컨테이너 모니터링은 컨테이너 환경을 지속적으로 운용하기 위한 필수적인 요소가 되었다.

컨테이너 모니터링은 컨테이너화된 인프라의 전체적인 상태를 실시간으로 확인할 수 있게 하며, 성능 문제의 근본적인 원인을 파악하고 개선할 수 있도록 한다. 또한 수집된 데이터의 시각화, 경고 자동화 등의 사용자 편의성을 위한 기능을 통해 효율적으로 컨테이너를 운용할 수 있도록 한다.

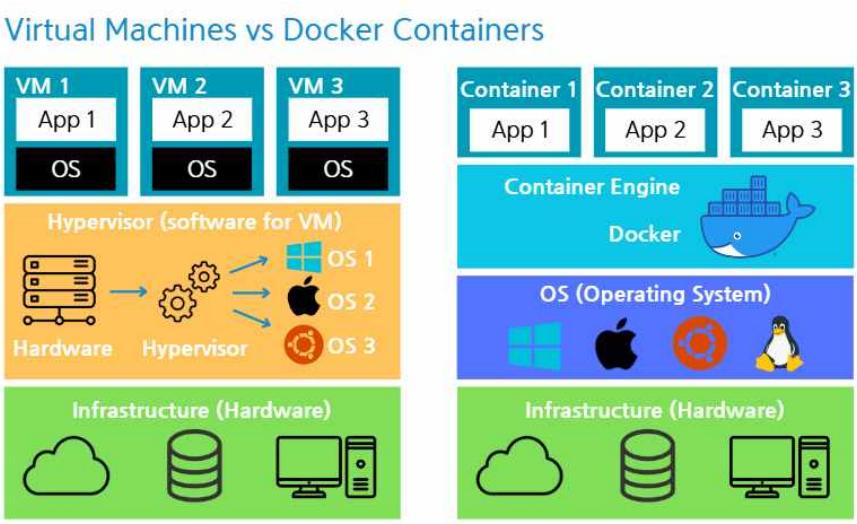
본 논문은 리눅스 환경에서 구동 중인 Docker 컨테이너의 메트릭을 cAdvisor를 통해 수집하고 수집한 데이터를 Prometheus의 시계열 데이터베이스에 저장시킨 후, 이를 Grafana를 통해 시각화시킴으로써 데이터를 효과적으로 분석하고 모니터링할 수 있는 시스템을 구현하는 것을 제시하고자 한다.

제2장 논문의 이론적 배경

2.1 Docker의 배경

Docker는 컨테이너 기반 가상화 플랫폼으로, 응용 프로그램과 그 종속성을 격리된 환경인 컨테이너로 패키징하여 실행하는 기술이다. 이를 통해 응용 프로그램을 서로 다른 환경에서도 일관되게 실행할 수 있고, 개발 환경과 운영 환경 사이의 차이로 인한 문제를 줄일 수 있다. Docker 컨테이너는 가볍고 빠르며 확장성이 좋아서 개발 및 배포 프로세스를 간소화하는 데 사용된다.

Docker Compose는 여러 개이 도커 컨테이너를 정의하고 실행하기 위한 도구로, 하나의 설정 파일로 여러 개의 컨테이너를 관리하고, 컨테이너 간의 네트워크 및 종속성을 설정하는 데 사용된다. 주로 복잡한 응용 프로그램이 여러 컴포넌트로 구성되어 있을 때 사용한다.



[그림 2-1] 가상 머신과 Docker 컨테이너의 차이 비교

컨테이너 기술이 아닌 기존의 가상화 방식은 주로 OS를 가상화했다. VMware, VirtualBox와 같은 가상 머신은 호스트 OS 위에 게스트 OS전체를 가상화하여 사용하는 방식으로 도커의 컨테이너 방식과는 큰 차이가 있다. 가상 머신은 [그림 2-1]과

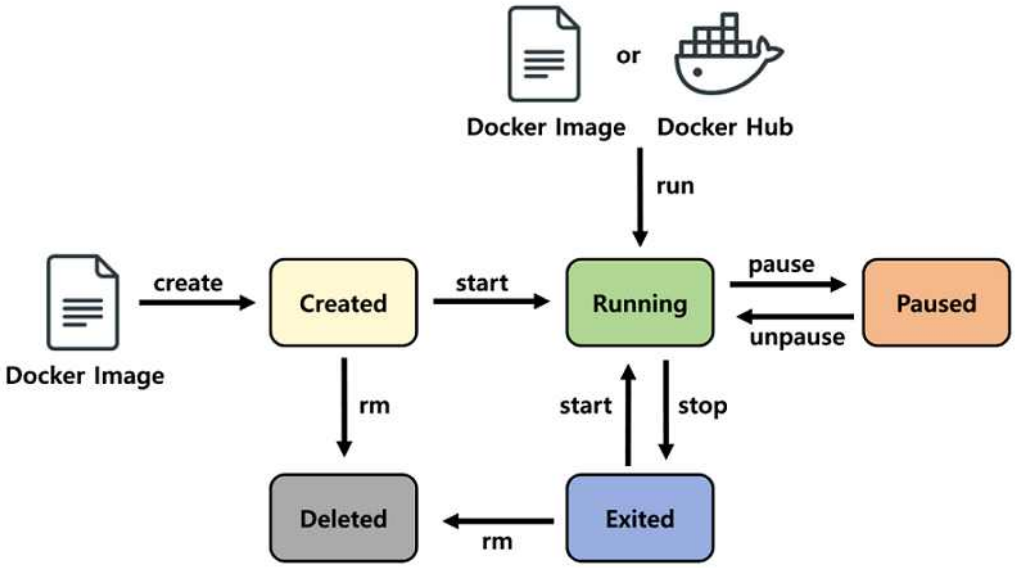
같이 호스트 운영체제 위에 가상화된 하드웨어 계층을 생성하고, 각 가상 머신은 독립된 운영체제, 커널, 드라이버 등을 가진다. 이로 인해 무겁고 자원 소비율이 높으며 운영체제의 부팅 과정이 필요하므로 실행 시 시간이 오래 걸린다. 반면에 Docker 컨테이너는 [그림 2-1]과 같이 컨테이너 엔진을 통해 격리된 컨테이너 환경을 생성한다. 이는 이미지와 컨테이너 레이어를 사용하여 빠르게 생성할 수 있으며, 실행 속도 또한 매우 빠르다. [4]

Docker 컨테이너를 생성하고 실행시키기 위해서는 Docker Image가 필요하다. Docker Image를 생성하려면 먼저 Dockerfile을 작성해야 한다. Dockerfile은 <표 2-1>과 같은 각 명령어 섹션을 통해 이미지의 구성 요소를 정의한다. 이러한 각 섹션에는 Docker Image에 필요한 내용과 명령어를 작성하여, 이미지 생성 과정을 체계적으로 구성할 수 있다.

<표 2-1> Dockerfile의 구성 요소

FROM	빌드할 베이스 이미지 지정
RUN	컨테이너에서 실행할 명령어 지정
ADD	컨테이너에서 배치할 파일이나 디렉토리 지정
CMD	컨테이너가 시작할 때 실행할 명령어 지정
ENTRYPOINT	컨테이너가 시작할 때 실행할 명령어 지정 (CMD에서는 param 값을 대체할 수 있지만 ENTRYPOINT에서는 불가능)
LABEL	key-value 형식의 메타데이터를 이미지에 추가
ENV	LABEL과 동일하지만 메타데이터 대신 환경변수 설정
VOLUME	컨테이너 내의 특정 디렉토리 지정

Dockerfile은 확장자가 따로 없으며 dockerfile 혹은 Dockerfile이라는 이름으로 생성하면 Docker가 자동으로 인식한다. 또한 다른 이름으로 Dockerfile을 생성하면 docker build -f <파일명> 명령어를 통해 특정 Dockerfile을 지정하여 빌드할 수 있다.



[그림 2-2] Docker 컨테이너 생성 및 실행 과정

(출처: https://tech.cloudmt.co.kr/2022/06/29/도커와-컨테이너의-이해-1-3-컨테이너-사용법/)

<표 2-2> Docker 컨테이너 명령어

docker create	컨테이너 생성
docker run	컨테이너 생성 및 자동 실행
docker start	정지 상태의 컨테이너 실행
docker stop	실행 중인 컨테이너 종료
docker pause	실행 중인 컨테이너 일시 정지
docker unpause	정지된 컨테이너 재개
docker rm	종료 상태 컨테이너 삭제

Dockerfile을 빌드하여 생성된 Docker Image는 [그림 2-2]와 같이 docker run 명령어 또는 docker create 명령어로 컨테이너화시킬 수 있다. 또한, 이렇게 생성된 컨테이너는 다양한 명령어(<표 2-2> 참조)를 통해 관리하고 조작할 수 있다.

2.2 cAdvisor의 배경

[그림 2-3] cAdvisor 로고

(출처: https://hub.docker.com/r/google/cadvisor)

cAdvisor는 Container Advisor의 약자로, Google에서 컨테이너를 모니터링하기 위해 개발한 오픈소스 도구이다. [5] cAdvisor는 Docker 컨테이너를 포함해 다양한 컨테이너 유형을 자동으로 검색하고 CPU 및 메모리 사용, 파일 시스템 및 네트워크 통계와 같은 컨테이너 기반 메트릭을 수집한다. 수집된 데이터를 Prometheus, Elasticsearch, InfluxDB와 같은 저장소로 전송할 수 있으며, 내장 웹 UI를 통해 실시간 메트릭을 시각화한다. cAdvisor는 이와 같은 기능들로 사용에 대한 편리함, 유연성 그리고 컨테이너 모니터링에 있어 다양한 요구 사항을 충족하기 때문에 컨테이너 모니터링의 대표적인 도구로 자리 잡았다. [6]

2.3 Prometheus의 배경

Prometheus는 SoundCloud에서 개발된 오픈 소스로 시스템 모니터링 및 경고 도구이다. 2012년 출시 이후 Prometheus는 많은 기업과 조직이 채택해 사용되어 졌으며, 개발자와 사용자 간의 커뮤니티를 형성해 왔다. 현재는 특정 기업에 종속되지 않고 독립적인 오픈 소스 프로젝트로 운영되고 있다. 이러한 독립성을 강조하고 프로젝트의 관리 구조를 명확히 하기 위해, Prometheus는 2016년 Kubernetes에 이어 두 번째로 클라우드 네이티브 컴퓨팅 재단에 합류하였다.

[그림 2-4] Prometheus 로고

(출처: https://en.wikipedia.org/wiki/Prometheus_(software))

Prometheus는 수집한 메트릭을 시간과 함께 기록하며, 추가 정보를 키-값 쌍으로 저장해 다차원적으로 데이터를 분석할 수 있도록 한다. 또한, Prometheus는 특정 상황이나 조건에 맞는 데이터를 효과적으로 조회할 수 있는 강력한 쿼리 언어를 제공한다. 데이터 수집은 주로 Pull 방식을 사용하며, 모니터링 대상 서버나 애플리케이션을 자동으로 탐지하는 기능을 통해 설정된 목록이나 서비스 디스커버리 기능으로 모니터링 대상을 효율적으로 관리할 수 있다. 수집된 데이터는 다양한 형태로 시각화되며, 대시보드를 통해 실시간 상태를 편리하게 파악할 수 있다. [7]

2.4 Grafana의 배경

Grafana는 Grafana Labs에서 개발한 오픈 소스 데이터 시각화 플랫폼으로, 사용자가 데이터를 차트와 그래프 형태로 하나의 대시보드(또는 여러 대시보드)에서 통합하여 쉽게 해석하고 이해할 수 있도록 도와준다. Grafana는 다양한 저장소에 있는 정보와 메트릭을 쿼리하고 알림을 설정할 수 있으며, 이를 통해 전통적인 서버 환경, Kubernetes 클러스터, 다양한 클라우드 서비스 등 어디에서든 데이터를 조회할 수 있다. 이를 통해 데이터를 쉽게 분석하고, 추세와 비일관성을 파악하며, 궁극적으로 프로세스를 더 효율적으로 만들 수 있다. Grafana는 데이터를 소수의 사람에게만 국한하지 않고 조직 전체에서 접근할 수 있어야 한다는 오픈 원칙에 기반을 두고 설계되

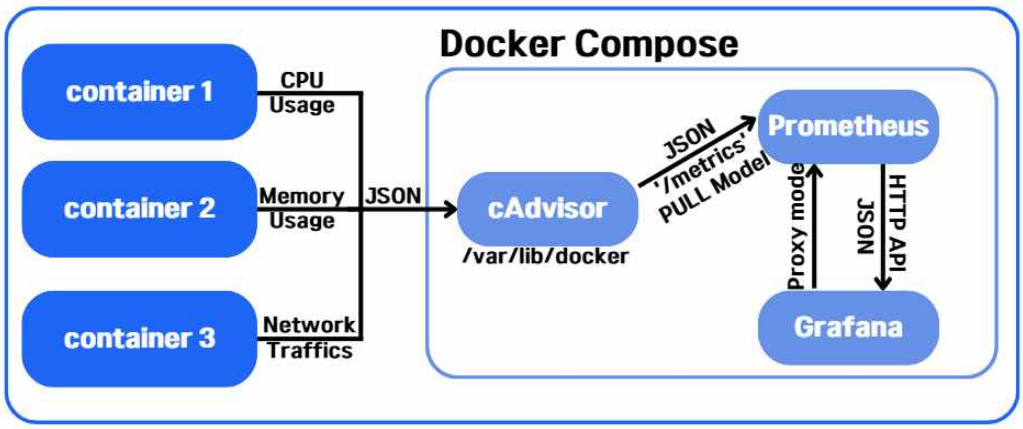
었다. [8]

[그림 2-5] Grafana 로고

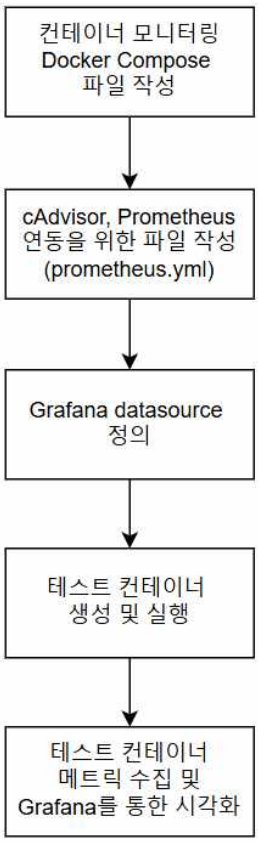
(출처: https://grafana.com/grafana/)

제3장 연구내용

3.1 시스템 작동 설계



[그림 3-1] 작동 요약



[그림 3-2] 컨테이너 모니터링 시스템 설계 순서도

Ubuntu 24.04 환경에서 Docker와 Docker compose를 설치한다. Docker compose는 단일 서버에서 다수의 컨테이너를 하나의 서비스로 정의하고 실행시키기 위한 도구이다. 파일 작성은 Dockerfile 작성 방식과 유사하며 yaml파일로 작성한다. cAdvisor, Prometheus, Grafana가 서로 데이터를 주고 받을 수 있도록 정의한 후, 이를 Docker compose를 통해 하나의 컨테이너로 묶어 실행시킨다. cAdvisor가 실행 중인 다른 Docker 컨테이너의 메트릭을 수집하고, 이 메트릭을 Prometheus가 스크랩하여 자체 시계열 데이터베이스에 저장시킨다. 이후 저장된 데이터는 Grafana를 이용하여 시각화시킨다. 전체적인 작동 요약은 [그림 3-1]과 같으며, [그림 3-2]와 같은 순서로 컨테이너 모니터링 시스템 구현을 진행할 예정이다.

본 연구에서는 구현과 사용의 편의성을 고려하여 Docker compose로 cAdvisor, Prometheus, Grafana 각각의 컨테이너를 한 번에 생성하였지만, 필요한 경우 Dockerfile로 작성하여 개별의 컨테이너를 생성한 후 서로 연동가능하다.

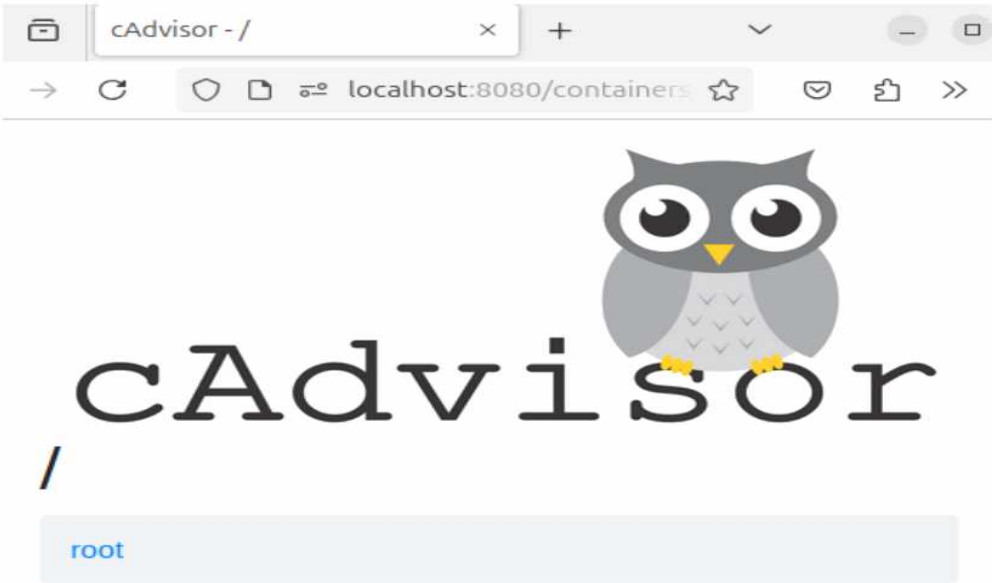
3.2 Docker compose 파일 cAdvisor 작성

Docker와 Docker compose를 설치했다면 cAdvisor, Prometheus, Grafana의 컨테이너를 생성할 Docker compose 파일을 작성한다. Docker compose 파일은 상단에 사용할 Docker compose의 버전을 작성하며, 이후 services 섹션에 내가 생성할 컨테이너들의 정보와 설정을 입력해준다. 다음 <표 3-1>은 생성할 cAdvisor 컨테이너의 정보를 작성한 Docker compose 파일 내용이다.

<표 3-1>과 같이 생성할 컨테이너의 이름을 cadvisor로 작성한다. 작성한 컨테이너의 이름은 Docker에서 컨테이너를 식별하는 데에 사용된다. 다음은 컨테이너를 생성하는 데에 사용할 이미지이다. cAdvisor의 이미지는 Google Container Registry(gcr.io)[9]에서 가져온다. 네트워크와 포트를 설정한다. 네트워크는 컨테이너가 호스트의 네트워크 설정을 직접 사용할 수 있도록 “host”로 설정하며, 포트는 cAdvisor 컨테이너의 포트 8080을 호스트의 포트 8080에 바인딩하여 호스트 브라우저에서 http://<호스트 IP>:8080으로 접속하면 cAdvisor의 UI에 다음 [그림 3-3]과 같이 접근할 수 있다.

<표 3-1> cAdvisor 작성 내용(Docker compose 파일 내용 중 일부)

```
cadvisor:
  container_name: cadvisor
  image: gcr.io/cadvisor/cadvisor:latest
  network_mode: "host"
  ports:
    - "8080:8080"
  volumes:
    - "/:/rootfs"
    - "/var/run:/var/run"
    - "/sys:/sys"
    - "/var/lib/docker/:/var/lib/docker"
    - "/dev/disk/:/dev/disk"
  privileged: true
  devices:
    - "/dev/kmsg"
```



[그림 3-3] 브라우저에서 접속한 cAdvisor UI

다음은 volumes 섹션이다. 이 섹션은 컨테이너가 호스트의 파일 시스템에 접근할 수 있도록 바인딩한 볼륨들을 정의한다. “/:/rootfs”는 cAdvisor가 호스트의 최상위 파일 시스템(/)을 /rootfs 경로로 마운트한다. 이에 따라 cAdvisor가 호스트 전체 파일 시스템을 모니터링할 수 있게 된다. “/var/run:/var/run”은 호스트의 /var/run 디렉터리를 cAdvisor가 접근할 수 있게 하여 이를 통해 컨테이너 정보를 수집할 수 있게 한다. “/sys:/sys” 는 호스트의 /sys 디렉터리를 컨테이너 내에서 /sys 경로로 마운트하여 cAdvisor가 호스트 시스템 정보를 수집할 수 있도록 한다. 여기에는 CPU,

메모리 등 시스템의 저수준 하드웨어 정보를 포함하고 있다. “/var/lib/docker/:/var/lib/docker”는 Docker의 메타데이터 및 컨테이너 파일들이 저장된 디렉터리를 마운트하여 cAdvisor가 Docker 관련 정보를 수집할 수 있게 한다. “/dev/disk/:/dev/disk”는 호스트의 디스크 정보를 포함하는 /dev/disk 디렉터리를 마운트하여 디스크 사용량 등의 메트릭을 모니터링할 수 있게 한다.

다음은 privileged 섹션이다. 이 설정에서 컨테이너에 루트 권한을 부여하여 더 많은 시스템 리소스와 하드웨어에 접근할 수 있게 한다. 이어서 devices 섹션에서는 호스트의 커널 로그(/dev/kmsg)파일을 컨테이너에 노출시켜 cAdvisor가 시스템 로그에 접근하여 추가적인 메트릭을 수집하거나 분석할 수 있도록 한다.

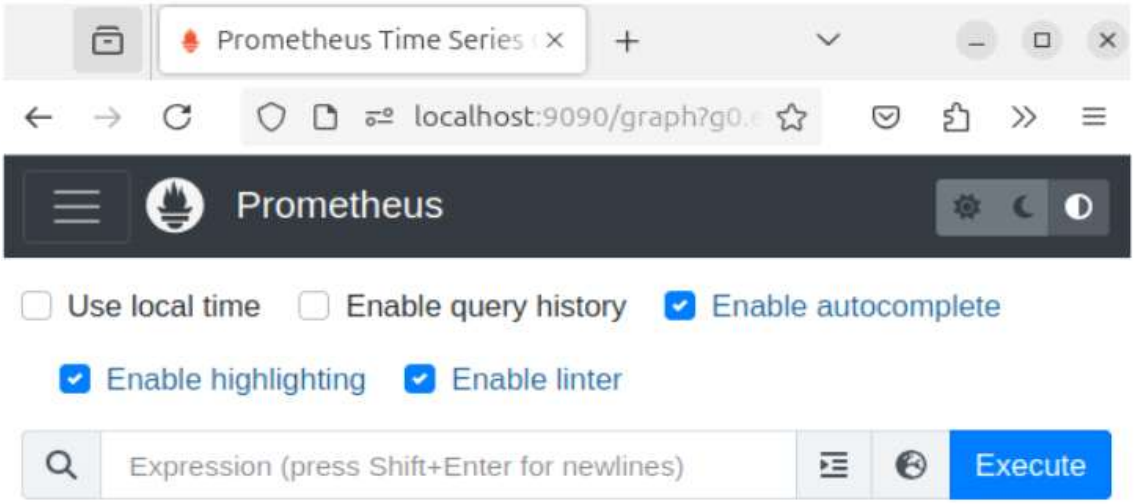
3.3 Docker compose 파일 Prometheus 작성

cAdvisor 컨테이너 내용을 작성한 후, 다음으로 Prometheus 컨테이너 설정을 작성한다. Prometheus는 시계열 데이터를 수집하고 관리하며 모니터링 및 알림 시스템을 제공하는 도구이다. 본 연구에서는 Prometheus를 사용하여 cAdvisor가 수집한 데이터를 스크랩해 자체 시계열 데이터베이스에 저장하고, 이를 Grafana와 연동해 시각화할 계획이다. Docker compose 파일에 작성한 Prometheus 컨테이너의 설정은 다음 <표 3-2>와 같다.

<표 3-2> Prometheus 작성 내용(Docker compose 파일 내용 중 일부)

```
prometheus:
    container_name: prometheus
    image: prom/prometheus:latest
    network_mode: "host"
    ports:
      - "9090:9090"
    volumes:
      - "./prometheus.yml:/etc/prometheus/prometheus.yml"
    privileged: true
    depends_on:
      - cadvisor
```

컨테이너의 이름은 prometheus로 작성한다. 다음은 Prometheus 컨테이너에서 실행시킬 이미지를 가져온다. 이미지는 Docker Hub의 prom/prometheus 저장소[10]에서 가져와 사용한다. 네트워크는 “host”로 설정하며, 포트 또한 Prometheus 컨테이너의 내부 포트 9090을 호스트의 9090 포트에 바인딩하여 Prometheus의 웹 UI와 API를 호스트 시스템의 브라우저에서 http://<호스트 IP>:9090으로 다음 [그림 3-3]과 같이 접근할 수 있도록 한다.



[그림 3-4] 브라우저에서 접속한 Prometheus UI

volumes 섹션에는 "./prometheus.yml:/etc/prometheus/prometheus.yml"를 통해 호스트의 현재 디렉터리에 있는 prometheus.yml 파일을 컨테이너 내부의 /ect/prometheus/prometheus.yml 경로에 마운트한다. prometheus.yml 파일은 데이터 수집 간격과 수집 대상을 설정한다. 이를 통해 Prometheus가 cAdvisor의 성능 및 리소스 사용량과 cAdvisor가 수집한 데이터를 스크랩할 수 있게 된다. 이에 대한 prometheus.yml 파일의 내용은 다음 <표 3-3>에 제시한다.

depends_on 섹션은 cAdvisor가 수집한 데이터를 Prometheus가 스크랩해야 하므로 cAdvisor 컨테이너가 실행된 후에 Prometheus 컨테이너를 실행하도록 설정한다.

<표 3-3> prometheus.yml 작성

```
global:
  scrape_interval: 15s #15초 간격으로 스크래핑
  evaluation_interval: 15s #15초 간격으로 정의된 작업 수행

scrape_configs:
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]

  - job_name: "cadvisor"
    static_configs:
      - targets: ["localhost:8080"]
```

<표 3-3>을 보면 scrape_interval과 evaluation_interval을 15초로 설정하여, Prometheus가 15초마다 cAdvisor 등의 데이터 소스에서 메트릭을 수집하고, 수집된 데이터에 기반해 규칙을 평가하여 정의된 작업을 수행하도록 한다.

scrape_configs 섹션은 Prometheus가 데이터를 수집할 대상을 정의하는 부분이다. job_name: "prometheus"는 Prometheus 자체의 메트릭을 수집하도록 설정하는 것으로, static_configs에 고정된 수집 대상을 지정한다. Prometheus 서버를 localhost:9090에서 실행할 예정이므로 이 경로에서 메트릭을 수집하도록 설정한다. 다음으로 job_name: "cadvisor"는 cAdvisor의 메트릭을 수집하는 설정으로, 앞의 Prometheus 설정과 동일하게 cAdvisor가 localhost:8080에서 실행될 예정이므로 static_configs의 주소를 localhost:8080으로 지정한다.

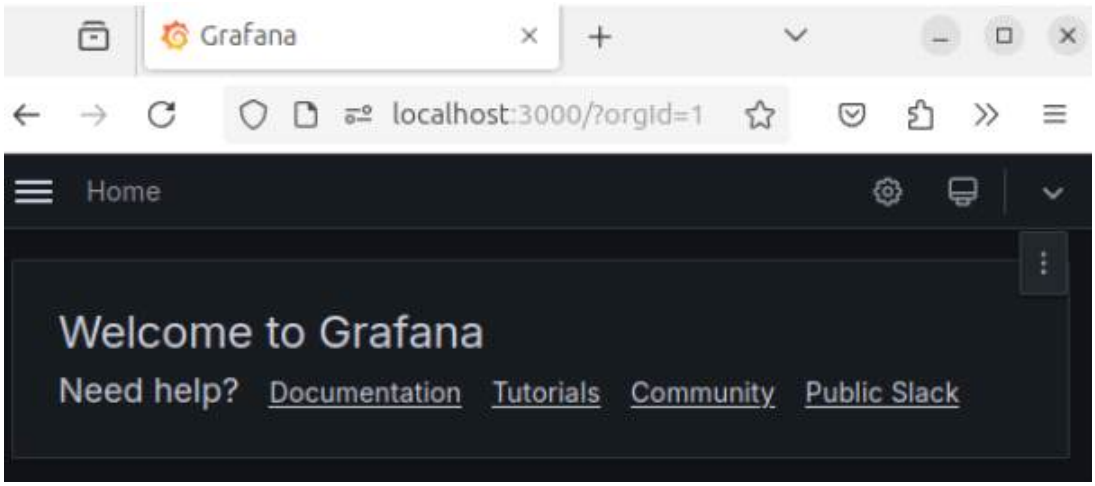
3.4 Docker compose 파일 Grafana 작성

Prometheus는 시계열 데이터를 수집하고 관리하는 데 강점이 있지만, 이 데이터를 텍스트나 숫자로만 확인하는 것에 한계가 있다. 이에 Grafana를 통해 Prometheus가 수집한 데이터를 시각화하여 직관적으로 이해하고, 효율적으로 분석할 수 있도록 한다. 다음 <표 3-4>는 Prometheus에 저장된 시계열 데이터들을 시각화시키기 위한 Grafana 컨테이너의 내용이다.

<표 3-4> Grafana 작성 내용(Docker compose 파일 내용 중 일부)

```
grafana:
    container_name: grafana
    image: grafana/grafana:latest
    network_mode: "host"
    ports:
      - "3000:3000"
    environment:
      - GF_PATHS_PROVISIONING=/etc/grafana/provisioning
      - DS_PROMETHEUS=prometheus
    volumes:
      - "grafana-data:/var/lib/grafana"
      - "./datasources.yml:/etc/grafana/provisioning/datasources/datasources.yml"
    privileged: true
    depends_on:
      - prometheus
```

컨테이너의 이름은 grafana로 작성한다. Grafana 컨테이너에서 실행시킬 이미지는 Docker Hub의 grafana/grafana 저장소[11]에서 가져와 사용한다. 네트워크는 “host”로 설정하며, 포트는 Grafana의 기본 포트인 3000 포트와 호스트의 3000 포트를 연결하여 브라우저에서 http://<호스트 IP>:3000으로 접속하면 다음 [그림 3-5]와 같이 Grafana 대시보드에 접근할 수 있도록 한다.



[그림 3-5] 브라우저에서 접속한 Grafana UI

environment 섹션에서 환경변수를 설정한다. GF_PATHS_PROVISIONING=/etc/grafana/provisioning은 Grafana의 데이터 소스, 대시보드 등의 자동 설정 파일들이 저장된 디렉터리를 가리킨다. DS_PROMETHEUS=prometheus는 prometheus라는 이름으로 Prometheus를 기본 데이터 소스로 설정한다. 이 환경변수를 통해 Prometheus에서 데이터를 가져오도록 한다.

volumes 섹션에서 "grafana-data:/var/lib/grafana"는 Grafana 데이터를 저장하는 볼륨이다, 이 경로를 통해 Grafana의 설정과 데이터가 저장되며, 컨테이너가 재시작되더라도 설정값을 유지하도록 한다. "./datasources.yml:/etc/grafana/provisioning/datasources/datasources.yml"은 호스트의 datasources.yml 파일을 Grafana의 프로비저닝 디렉터리에 마운트한다. 이 파일을 통해 Grafana에서 사용할 데이터 소스를 미리 설정할 수 있다. datasources.yml 파일의 내용은 다음 <표 3-5>에서 제시한다.

Depends_on 섹션에서는 Grafana가 Prometheus의 데이터를 시각화하기 위해 Prometheus 컨테이너가 실행된 후에 Grafana 컨테이너가 실행되도록 설정하면 된다.

<표 3-5> datasources.yml 작성

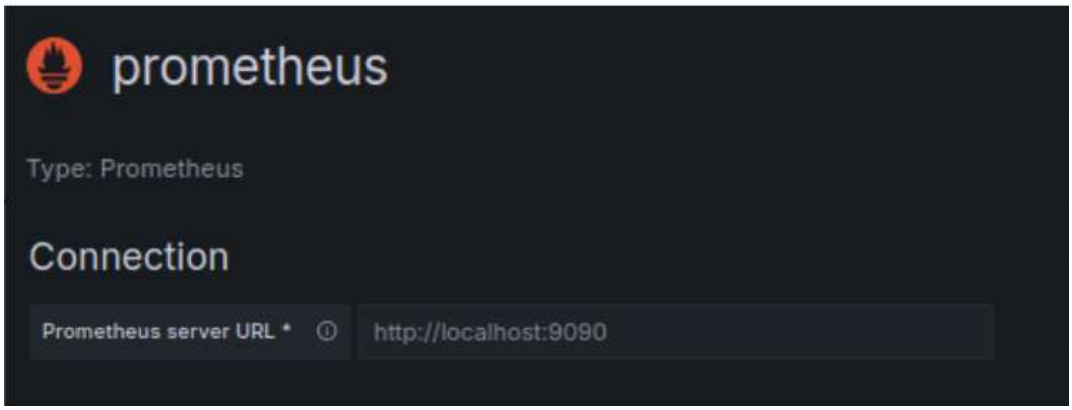
```
datasources:
  - name: prometheus
    type: prometheus
    url: http://localhost:9090
    access: proxy
    isDefault: true
```

datasouces.yml 파일에는 Grafana가 사용할 데이터 소스로 Prometheus를 지정하고 있으며, 데이터를 가져올 위치와 proxy 모드를 통해 Grafana 서버가 간접적으로 Prometheus에 데이터를 요청할 수 있게 하는 내용을 담고 있다.

3.5 Grafana와의 연동

작성한 Docker Compose 파일을 docker-compose up 명령어로 실행하면 설정된 cAdvisor, Prometheus, Grafana 컨테이너가 각각 생성되어 동시에 실행된다. 이 명령어는 Docker Compose 파일에 정의된 각 서비스를 자동으로 생성하고 연결하여, 컨테이너들이 독립적이면서도 연동된 상태로 작동할 수 있도록 한다.

Grafana에 접속하기 위해 주소창에 http://<호스트 IP>:3000을 입력한다. Grafana에서 datasource로 Prometheus를 설정하기 위해 Datasources에 Prometheus를 선택하고 다음 [그림 3-6]과 같이 Prometheus server URL을 prometheus 컨테이너에서 설정한 것과 같은 http://localhost:9090으로 입력한다.

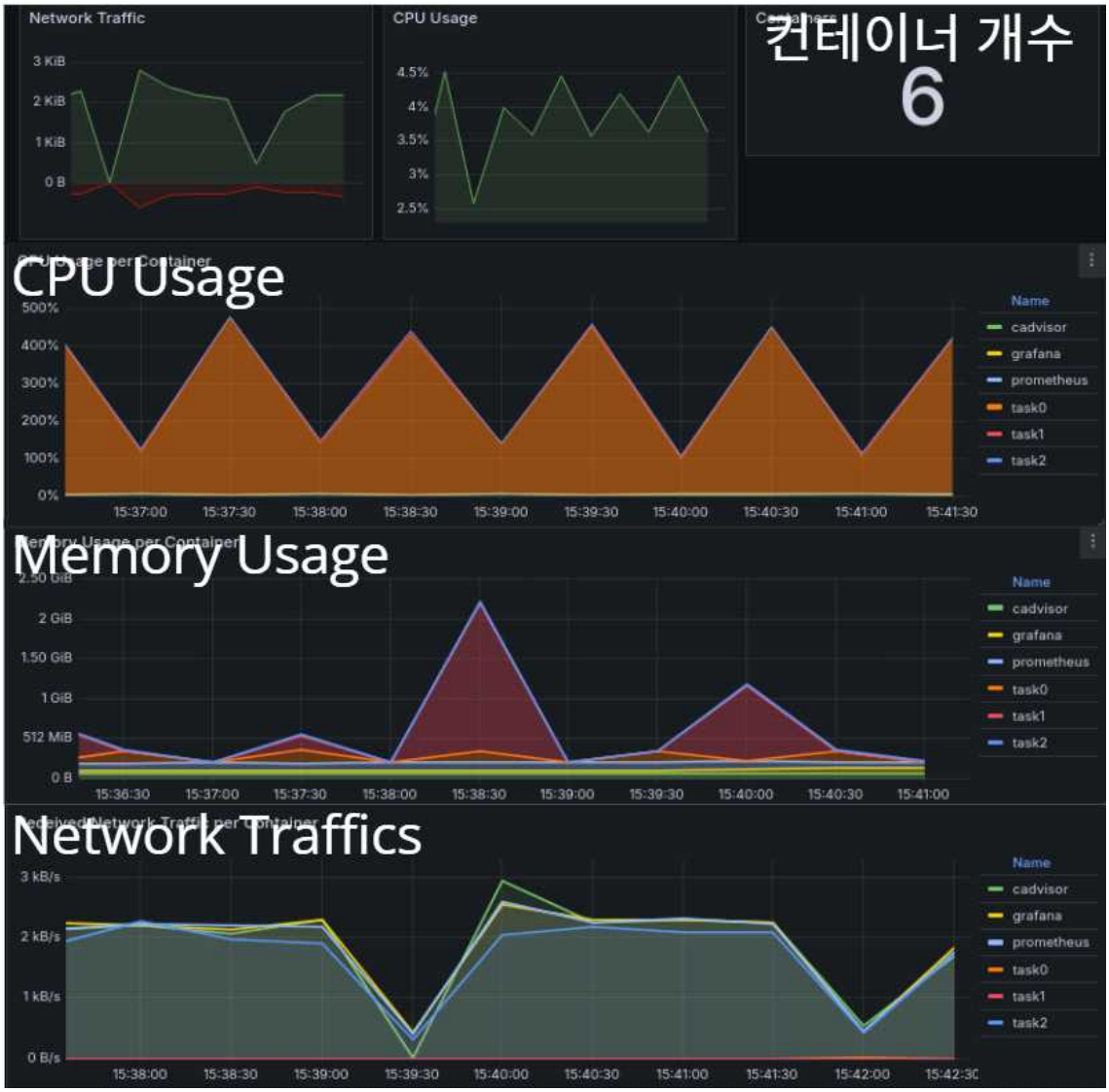


[그림 3-6] Grafana UI에서의 Datasource 설정

다음은 Prometheus에 요청한 데이터를 시각화시킬 Grafana의 dashboard를 설정한다. Import dashboard에서 Grafana dashboard ID를 입력하여 사용할 dashboard를 불러온다. 본 연구에서는 cAdvisor에서 오픈 소스로 제공하는 dashboard를 불러와 사용하였지만 따로 JSON 파일을 업로드하거나 코드를 입력하는 공간에 JSON 코드를 입력하여 dashboard를 구성해도 무방하다. [12]

구성한 dashboard를 통해 Docker 컨테이너에서 수집한 데이터를 그래프로 표현하고 분석할 수 있다. 각 그래프는 서로 다른 메트릭과 컨테이너로 분류되어 데이터의 흐름과 패턴을 쉽게 파악할 수 있다.

구현한 컨테이너 모니터링 시스템을 테스트하기 위해 단일 컨테이너들을 생성하였다. task0, task1, task2로 총 3개의 컨테이너이며, 각각 task0는 CPU에 과부하를 주는 코드, task1는 배열을 생성하여 일시적으로 메모리 사용량을 증가시키는 코드, task2는 Network traffic을 유발시키는 코드를 내포하고 있다.



[그림 3-7] 다수의 컨테이너들을 실행시켰을 때의 dashboard 모습

생성한 다수의 컨테이너들을 동시에 실행시켰을 때, 각 컨테이너의 CPU usage, Network traffic, Memory usage가 Grafana에서 시각화되어 실시간으로 보여지고 있는 모습으로 장기간의 그래프 변화 추이를 한눈에 파악할 수 있다.

제4장 결 론

본 논문에서 구현한 cAdvisor, Prometheus, Grafana를 활용한 Docker 컨테이너 모니터링 시스템은 컨테이너가 생성하는 메트릭들을 효율적으로 수집하고 이를 실시간으로 모니터링함으로써 컨테이너 환경에서 구동되는 애플리케이션의 안정성을 높일 수 있었다.첫째, cAdvisor를 통해 실시간으로 Docker 컨테이너의 CPU, 메모리, 네트워크, 디스크 사용량 등의 다양한 메트릭 데이터를 수집할 수 있었으며, 이를 통해 애플리케이션의 상태를 지속적으로 모니터링하는 것이 가능해졌다. 둘째, Prometheus의 시계열 데이터베이스는 수집한 데이터를 효율적으로 저장하고 관리함으로써, 장기적인 데이터를 바탕으로 컨테이너의 성능 변화를 분석하고 예측할 수 있도록 도와주었다. 셋째, Grafana는 수집된 데이터를 시각화 시킴으로써, 메트릭 정보를 한눈에 파악하고 문제의 근본적인 원인을 쉽게 찾을 수 있는 환경을 제공하였다.

위와 같이 구현된 시스템을 통해 Docker 컨테이너 환경에서 발생할 수 있는 성능 문제를 사전에 식별하고 해결할 수 있으며, 시스템 자원 소모를 관리함으로써 지속적인 애플리케이션의 안정성을 확보할 수 있을 것이다.

추후 연구를 통해 Prometheus와 Grafana를 활용하여 컨테이너가 허용 범위를 초과하는 리소스를 사용하거나 컨테이너가 자동으로 중지되는 상황에서 경고 알림을 제공하는 기능을 추가할 계획이다. 이를 통해 사용자가 상황을 더 빠르게 상황을 인지하고 조치를 취할 수 있도록 할 것이다. 또한, 이러한 모니터링 시스템을 휴대폰 애플리케이션으로 개발하여 사용자가 실시간으로 편리하게 시스템 상태를 확인하고 관리할 수 있도록 개선할 예정이다.

참 고 문 헌

[1] 컨테이너, https://www.redhat.com/ko/topics/containers

[2] 컨테이너 기술의 장점, https://www.openmaru.io/%EC%BB%A8%ED%85%8C%EC%9D%B4%EB%84%88-%EA%B8%B0%EC%88%A0-%EC%86%8C%EA%B0%9C/

[3] 컨테이너 기술 시장 성장 지표, https://www.verifiedmarketreports.com/ko/product/container-technology-market/

[4] Docker 개념, https://mvje.tistory.com/161

[5] cAdvisor 배경, https://github.com/google/cadvisor

[6] cAdvisor 특징, https://www.kubecost.com/kubernetes-devops-tools/cadvisor/

[7] Prometheus, https://prometheus.io/docs/introduction/overview/

[8] Grafana, https://www.redhat.com/en/topics/data-services/what-is-grafana

[9] cAdvisor 이미지, https://console.cloud.google.com/artifacts/docker/cadvisor/us/gcr.io/cadvisor

[10] Prometheus 이미지, https://hub.docker.com/r/prom/prometheus

[11] Grafana 이미지, https://hub.docker.com/r/grafana/grafana

[12] Grafana dashboard, https://grafana.com/grafana/dashboards/893-main/

부 록

<부록 1> Docker-Compose File

```
version: "3"

services:
  cadvisor:
    container_name: cadvisor
    image: gcr.io/cadvisor/cadvisor:latest
    network_mode: "host"
    ports:
      - "8080:8080"
    volumes:
      - "/:/rootfs"
      - "/var/run:/var/run"
      - "/sys:/sys"
      - "/var/lib/docker/:/var/lib/docker"
      - "/dev/disk/:/dev/disk"
    privileged: true
    devices:
      - "/dev/kmsg"

  prometheus:
      container_name: prometheus
      image: prom/prometheus:latest
      network_mode: "host"
      ports:
        - "9090:9090"
      volumes:
        - "./prometheus.yml:/etc/prometheus/prometheus.yml"
      privileged: true
      depends_on:
        - cadvisor

  grafana:
    container_name: grafana
    image: grafana/grafana:latest
    network_mode: "host"
    ports:
      - "3000:3000"
    environment:
      - GF_PATHS_PROVISIONING=/etc/grafana/provisioning
      - DS_PROMETHEUS=prometheus
    volumes:
      - "grafana-data:/var/lib/grafana"
      - "./datasources.yml:/etc/grafana/provisioning/datasources/datasources.yml"
    privileged: true
    depends_on:
      - prometheus

volumes:
  grafana-data:
```

<부록 2> prometheus.yml

```
global:
  scrape_interval: 15s #Scrape interval to every 15 seconds.
  evaluation_interval: 15s #Evaluate rules every 15 seconds.

scrape_configs:
  - job_name: "prometheus"
    # metrics_path defaults to '/metrics'
    static_configs:
      - targets: ["localhost:9090"]

  - job_name: "cadvisor"
    static_configs:
      - targets: ["localhost:8080"]
```

<부록 3> datasources.yml

```
datasources:
  - name: prometheus
    type: prometheus
    url: http://localhost:9090
    access: proxy
    isDefault: true
```

<부록 4> cpu_task.py

```
import multiprocessing
import time

def cpu_stress():
    # 계산 작업을 계속 반복하여 CPU 부하 발생
    while True:
        [x**2 for x in range(1000000)]

if __name__ == '__main__':
    cpu_cores = multiprocessing.cpu_count()
    print(f"Using {cpu_cores} CPU cores.")

    while True:
        processes = []
        # CPU 코어 수 만큼 프로세스 생성
        for _ in range(cpu_cores):
            p = multiprocessing.Process(target=cpu_stress)
            processes.append(p)
            p.start()
        # 30초 동안 부하를 발생
        time.sleep(30)
        # 부하 종료
        for p in processes:
            p.terminate()
        print("30초간 부하 발생 후 30초간 휴식합니다.")
        # 30초간 휴식
        time.sleep(30)
```

<부록 5> memory_task.py

```
import time
import numpy as np
def memory_intensive_task():
    print("Starting memory intensive task...")
    large_memory_usage = []
    for _ in range(10):
        # 50MB 정도의 배열을 만들어 메모리 사용
        large_memory_usage.append(np.random.rand(5000, 5000))
        print(f"Memory in use: {len(large_memory_usage) * 50} MB")
        time.sleep(1)  # 메모리 사용량 증가 후 대기

if __name__ == "__main__":
    while True:
        memory_intensive_task()
        print("Memory task completed, waiting for 30 seconds before restarting...")
        time.sleep(30)  # 30초 대기 후 작업 재실행
```

<부록 6> network_task.py

```
import time
import requests

def network_intensive_task():
    print("Starting network intensive task...")
    url = "https://jsonplaceholder.typicode.com/posts"  # 테스트용으로 사용할 API
    for _ in range(5):
        response = requests.get(url)
        print(f"Response status: {response.status_code}, Length: {len(response.content)}")
        time.sleep(1)  # 요청 사이 대기 시간

if __name__ == "__main__":
    while True:
        network_intensive_task()
        print("Task completed, waiting for 30 seconds before restarting...")
        time.sleep(30)  # 30초 대기 후 작업 재실행
```

ABSTRACT

Implementation of Docker Container Monitoring System Using cAdvisor

Lee, Woo Jin
Computer Engineering
Honam University

In modern times, with the advancement of technologies such as big data, artificial intelligence, and the Internet of Things, the complexity of applications is increasing, and there is a growing demand for accelerated development speed. Consequently, the container environment has gained attention as an efficient solution for handling repetitive tasks. Containers offer the advantages of speeding up development and deployment, reducing inconsistencies between environments, and minimizing resource consumption, making them highly efficient across various infrastructures. For these reasons, container technology has been applied in various industries, including healthcare, finance, and retail, and its market size is continuously growing.

As the importance of the container environment increases, container monitoring has become essential to identify and resolve performance issues or security incidents in advance. Through monitoring, it is possible to check the overall status in real time, identify the causes of performance issues, and efficiently manage containers using data visualization and automated alerting functions.

In this paper, a Docker container monitoring system was implemented using cAdvisor, Prometheus, and Grafana. Each tool plays the following role. cAdvisor collects various metrics such as CPU, memory, and network usage

of Docker containers. Prometheus periodically scrapes these metrics and stores them in a time-series database, providing a monitoring and alerting system. Grafana retrieves data stored in Prometheus and visualizes it, allowing users to intuitively understand the status of containers. During the process of building the Docker container monitoring system, Docker Compose was used to configure the three tools as separate containers, enabling automatic integration of data collection, storage, and visualization. Various experiments were conducted to verify that this monitoring system shows real-time resource usage of each container. For example, three containers, task0 (inducing CPU load), task1 (temporarily increasing memory usage), and task2 (causing network traffic), were executed simultaneously to observe how the resource usage changes for each container were visualized in Grafana. This demonstrates that it is possible to monitor the performance and stability of container-based applications and provide an environment that allows for quick responses to issues as they arise.

Keywords (5): Docker container, Monitoring, cAdvisor, Prometheus, Grafana