IoT 환경에서 도커 컨테이너의 서비스 재시작 시간에 관한 분석†,‡

황승현*, 강지훈*, 정광식** 유헌창*
*고려대학교 대학원 컴퓨터학과
**한국방송통신대학교 컴퓨터학과
e-mail : {somnus, k2j23h, yuhc}@korea.ac.kr, kchung0825@knou.ac.kr

Analysis of Service Restart Time on the Docker Container in IoT Environment

Seung-Hyun Hwang*, Ji-Hun Kang*, Kwang-Sik Chung**, Heon-Chang Yu*
*Dept. of Computer Science and Engineering, Korea University
**Dept. of Computer Science, Korea National Open University

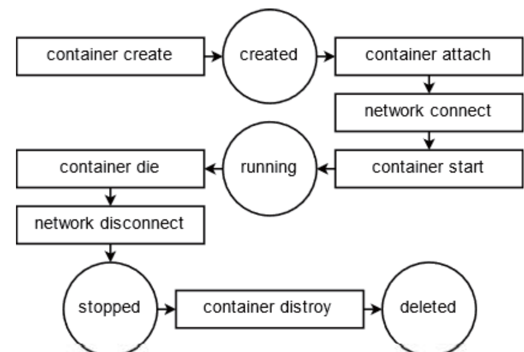
요 약

운영 체제 수준 가상화 기술의 결과물인 컨테이너는 경량화된 가상화 환경의 특징과 도커 프로젝트를 통한 손쉬운 패키지 관리와 배포를 특징으로 급격히 성장하였다. 이로써 컨테이너를 기반으로 한 IoT 클라우드 구성에 관한 연구가 진행되고 있지만, 한정된 자원과 성능을 가진 IoT 장치에서는 컨테이너의 가용성을 보장하기가 어렵다. 따라서 본 논문에서는 컨테이너의 서비스가 종료되고 다시 개시되기까지의 시간이 어떤 요인에 의하여 영향을 받는지 분석한다.

1. 서론

컨테이너는 운영 체제 수준 가상화(Operating Level Virtualization) 기술에 의해 호스트 운영체제 자원은 공유하면서 컨테이너 간 CPU, 메모리, 디스크, 그리고 네트워크 등의 자원이 격리된 사용자 공간을 일컫는다. 이는 하이퍼바이저(Hypervisor)를 통해 게스트 운영체제를 실행하는 방식에 비하여 서비스를 유지하는데 더 적은 자원을 소모하며 빠른 응답속도를 가진다는 장점이 있다. 또한 컨테이너는 의존관계에 있는 여러 레이어로 이루어진 이미지 형태로 관리되어 서비스는 하나의 이미지를 배포함으로써 이루어진다. 서비스의 업데이트는 기존의 이미지에 변경된 부분의 레이어를 올려 새로운 이미지를 만든다. 따라서 클라이언트가 이전 버전의 이미지를 가지고 있다면 새로운 레이어만 전송받아 클라이언트에서 업데이트된 이미지를 구성할 수 있다. 이러한 패키지 관리와 배포의 용이함, 응답 속도 그리고 컨테이너를 배치 자동화 오픈 소스 프로젝트인 도커(Docker)[1]와 함께 컨테이너에 대한 수요가 빠르게 증가하였다. 시장에서 도커가 주목받을 수 있었던 이유 중 하나는 컨테이너의 실행 환경을 Open Container Initiative(OCI) 기술 사양을 만족하는 runC 로 채택하고 여러 추상화 레이어를 통해 컨테이너가 다양한 환경에서 동작하게 할 수 있다는 것이다.

최근에는 IoT 장치에서 컨테이너를 기반으로 한 IoT 클라우드 구성에 관한 연구도 진행되고 있다.[2][3] 하지만 IoT 환경에서 사용되는 장치는 일반 서버 장치보다 컴퓨팅 성능이 떨어진다. 따라서 장치 또는 서비스에 문제가 생겨 컨테이너를 다시 시작해야 할 때 서비스가 재개되는데 적지 않은 시간이 걸린다. 이는 클라우드에서 갑작스러운 서비스 요청의 증가를 수용하기 어렵게 만든다.



(그림 1) 무상태 서비스를 수행하는 도커 컨테이너의 생명주기

따라서 본 논문에서는 도커 컨테이너를 이용하여 다른 컨테이너에 의해 특정 유형의 부하가 주어진 상황에서, 서비스 중인 컨테이너를 재시작하고 서비스가 재개되기까지 걸리는 시간을 측정하여 어떤 유형의 부하가 가장 크게 영향을 주는지 분석한다.

이 논문의 구성은 다음과 같다. 2 장에서 실험에 사용되는 도커 컨테이너의 생명주기에 대해 설명한다. 3 장은 실험을 진행한 후, 실험 결과에 대해 분석한다. 4 장에서는 본 논문과 관련된 기존의 연구에 대해 설명하고, 마지막으로 5 장에서 결론 및 향후 연구에 관해서 설명한다.

†본 연구는 과학기술정보통신부 및 정보통신기술진흥센터의 대학 ICT 연구센터 지원사업의 연구결과로 수행되었음 (IITP-2018-0-001405)
‡이 논문은 2018 년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2018-0-00480)

```
1  FROM debian:stretch-slim
2  RUN apt-get update\
3      && apt-get install -yg
4  ENTRYPOINT [ “stress-ng” ]
5  CMD [ “—help”]
```

(그림 2) stress-ng 를 실행하기위한 이미지의 Dockerfile

2. 도커 컨테이너의 생명주기

도커엔진은 컨테이너를 관리하기 위해 containerd 를 포함하고 있으며 containerd 에 의해 관리되지 않는 네트워크와 볼륨 등에 대한 관리가 구현되어있다. 본 논문은 컨테이너가 호스트 볼륨을 마운트 하지 않고 무상태(stateless) 작업을 수행하는 컨테이너를 가정하여 실험하며, 실험에 사용되는 도커 컨테이너의 생명주기(life cycle)는 (그림 1)과 같다. 컨테이너를 생성하면 사용되는 이미지의 레이어들은 읽기 전용으로 링크 되고 최상단에 읽기 및 쓰기가 가능한 레이어가 추가된다. 생성된 컨테이너는 프로세스를 실행하고 있지 않고 자원만 격리된 상태이다. 컨테이너가 생성되고, 컨테이너가 실행되기 전, 컨테이너의 표준 스트림이 사용자 터미널에 이어진다. 그리고 컨테이너가 네트워크 인터페이스와 연결되며 원격 프로시저 호출(Remote Procedure Call)을 통해 컨테이너에서 프로세스가 시작되며 실행 상태가 된다. 프로세스가 종료되고 health check 를 통해 종료가 감지되면 컨테이너는 네트워크 인터페이스와 연결을 끊고, 별도의 옵션이 없으면 실행 중인 프로세스 없이 컨테이너가 정지 상태에 남아있는다.

본 논문에서는 각 작업을 create, attach, connect, start, die, disconnect 그리고 destroy 라고 정의한다. 정지 상태의 컨테이너는 다시 시작될 수 있다. 하지만 클라우드 환경에서는 컨테이너가 다시 실패할 가능성을 줄이기 위해 실패가 발생했던 노드에서 다시 시작하지 않고, 다른 노드에 새로운 컨테이너를 만들어 서비스를 시작하기도 한다[4]. 실험에서도 그러한 정책을 가정하고 컨테이너의 종료 후 정지 상태에서 다시 시작하지 않는다. 실험은 기존의 컨테이너를 파기하고 새로운 컨테이너를 생성하여 실행 상태가 되기까지의 시간을 측정한다.

3. 실험

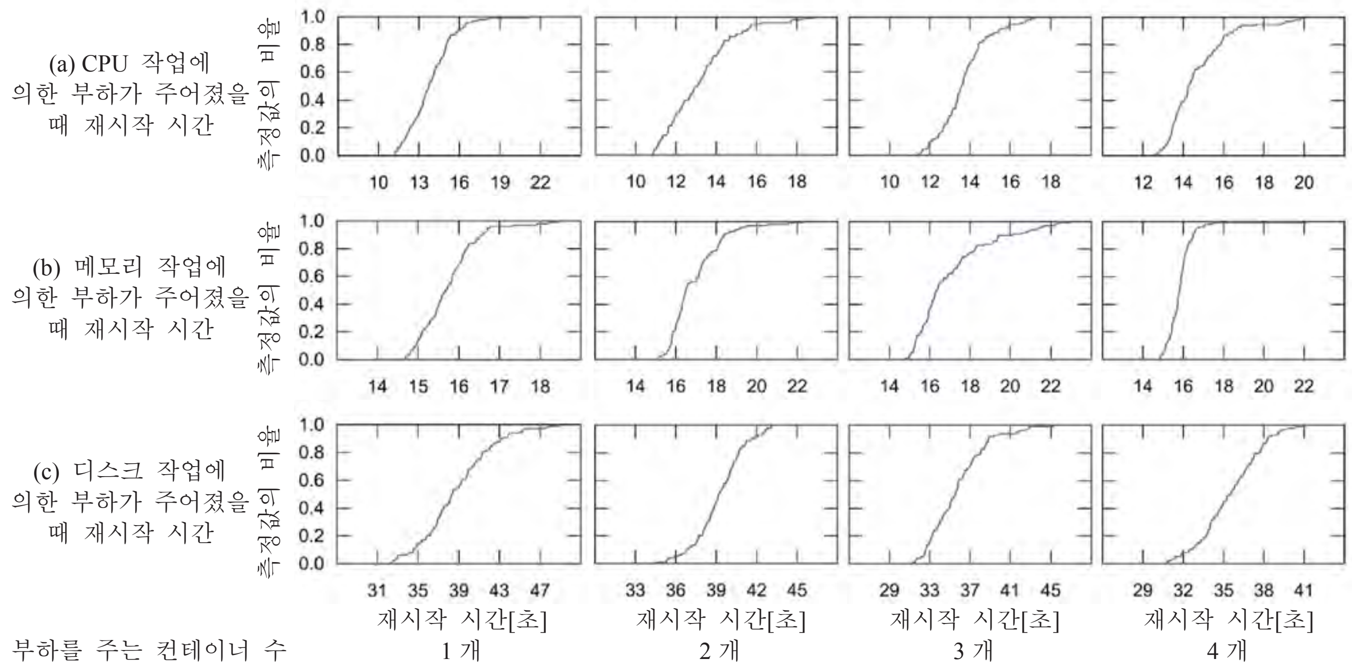
본 장의 실험에서는 다른 컨테이너에 의해 시스템에 다음 세 가지 유형: a) CPU 작업, b) 메모리 입출력 작업, c) 디스크 입출력 작업 중 하나의 유형에 의한 부하가 주어진 상황에서 서비스를 실행 중인 컨테이너를 종료하고 새로운 컨테이너에서 이전의 서비스가 다시 시작되기까지의 시간을 측정하였다. 성능의 기준을 제시하기 위해 부하를 주는 컨테이너가 없을 때 대한 실험도 진행하였다.

세 가지 유형의 부하는 각 실험마다 하나씩 주어지며 모든 코어를 사용하여 부하 작업을 실행하였다. 측정은 부하 작업을 실행하는 컨테이너가 1~4 개일 때, 실험 대상이 되는 컨테이너를 파기하고 생성하며 docker events 명령어를 통해 이벤트 정보를 수집했다. 동시에 실험 대상이 되는 컨테이너의 시작과 종료에 타임스탬프를 출력하여 서비스의 시작과 종료 시간을 측정하였다. 실험은 각각의 유형에 대해 100 회씩 진행하였다.

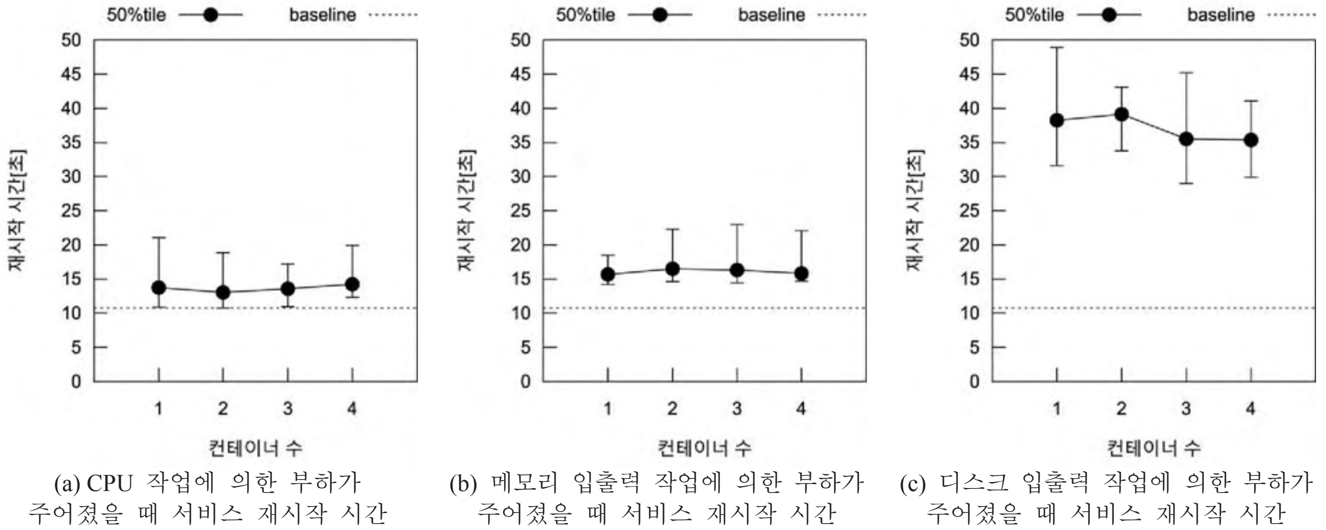
<표 1> 유형에 따른 stress-ng 옵션

유형	옵션
CPU	-c $(nproc)
메모리	-vm $(nproc) --vm-bytes 20m
디스크	-I $(nproc)

실험은 IoT 장치로 잘 알려진 Raspberry Pi2 에서 Debian stretch 운영체제를 사용하였다. 실험을 진행하면서 CPU scaling 에 의한 영향을 받을 수 있으므로 CPU governor 의 옵션을 performance 로 하여 주파수를 최대치고 고정시켰다. 실험에 사용된 Docker 는 18.06.0-ce 버전이며 실험에서 생성되는 컨테이너는 성능 격리와 관련된 어떠한 옵션도 없이 생성된다. 부하를 주는 작업을 실행하는 컨테이너는 Stress-ng[5]를 사용하였으며 해당 이미지의 빌드에 사용된 Dockerfile 은 (그림 2)와 같다. Stress-ng 를 실행할 때 주어진 명령어는 부하 유형에 따라 <표 **1**>과 같다. 실험 대상이 되는 컨테이너는 도커 허브의 공식 이미지 node:8-slim 를 사용하여 스크립트를 실행한다. 사용된 SD 카드의 성능은 45MB/s 읽기, 최소 10MB/s 의 데이터 전송률이다.



(그림 3) 시스템에 주어지는 부하 유형에 따른 컨테이너에서의 서비스 재시작 시간의 누적 분포 함수



(a) CPU 작업에 의한 부하가 주어졌을 때 서비스 재시작 시간

(b) 메모리 입출력 작업에 의한 부하가 주어졌을 때 서비스 재시작 시간

(c) 디스크 입출력 작업에 의한 부하가 주어졌을 때 서비스 재시작 시간

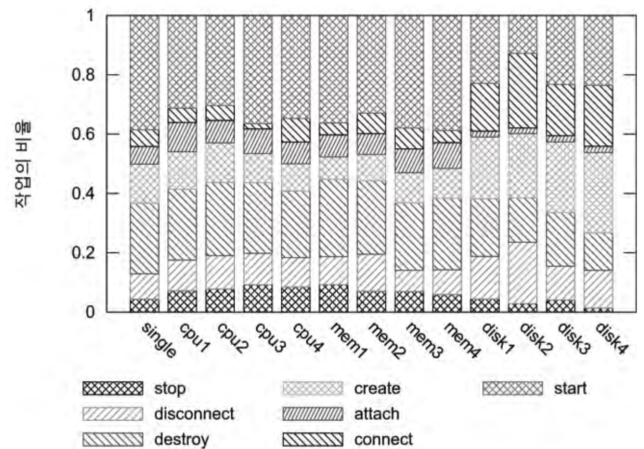
(그림 4) 시스템에 주어지는 부하 유형에 따른 컨테이너에서의 서비스 재시작 시간

서비스 재시작 시간의 측정 기준은 컨테이너에서 node 프로그램에 의해 시작 직후와 종료 직전에 출력되는 타임스탬프이다. 측정값은 전자와 후자의 차로 구해진다. 2 장에서 정의한 start 작업과 die 작업이 끝나는 시점을 기준으로 하지 않는 이유는 start 작업은 서비스는 시작되지 않고 프로세스만 실행된 시점에서 작업이 완료될 수 있고, die 작업은 서비스 프로세스가 종료된 후 시작되기 때문이다. (그림 3)은 부하를 주는 유형과 해당 유형의 부하를 주는 컨테이너의 수에 따라 수행한 실험에서 측정된 값을 누적 분포함수로 나타낸 것이다. (그림 4)는 측정값들의 중앙값이다. 오차 막대는 해당 측정값들의 최솟값과 최댓값을 나타낸다. 기준선은 부하를 주는 컨테이너가 없는 상황에서 실험하여 측정한 값의 중앙값으로, 10.788 이다.

모든 유형에서 부하가 없을 때 보다 성능이 저하되었지만, 컨테이너의 수가 늘어난다고 성능이 저하되지 않는 것을 확인할 수 있다. a) CPU 작업 유형의 각 중앙값의 평균은 13.65, b) 메모리 입출력 작업 유형에서의 평균은 16.077 로 기준값으로부터 각각 26.53%, 49.02% 만큼 성능이 저하되었다. c) 디스크 입출력 작업 유형에서의 평균은 37.072 로 기준값으로부터 243.64%만큼 성능이 저하되어 세 유형 중 c) 디스크 입출력 작업 유형에 의한 성능 저하가 가장 큰 것을 확인할 수 있다.

(그림 5)는 부하를 주는 작업 유형에 따른 각 실험의 측정값이 중앙값인 실험에서 2 장에 정의한 작업이 차지하는 비율을 나타낸 것이다. 실험 유형의 single 은 부하를 주는 다른 컨테이너가 없는 실험이고, cpu, mem 그리고 disk 는 각각 a) CPU, b) 메모리 입출력, 그리고 c) 디스크 입출력 작업에 의한 부하가 주어질 때의 실험이다. 숫자는 동시에 부하를 주는 컨테이너의 수를 의미한다.

부하가 없을 때, 그리고 a) CPU 와 b) 메모리 입출력 작업에 의한 부하가 주어질 때는 start 작업이 전체 작업의 30.35%~38.78%로 가장 많은 시간을 소모했다. 반면에, c) 디스크 입출력 작업에 의한 부하가 주어지자 create 작업이 전체 작업의 20.86%~27.11%로 증가하며 가장 많은 시간을 소모하게 되었다. 또한 connect 작업과 disconnect 작업 비율 평균이 각각 4.87%, 9.67%에서 19.47%, 14.41%로 증가하였다. Stop 작업과 attach 작업은 모든 유형의 실험에서 각각 12.21 ±2.3, 10.47±3.03 으로 측정되었는데, 다른 작업이 c) 디스크 입출력 작업에 의한 부하 유형의 실험에서 최소 34.49% 만큼 증가한 것에 비해 비교적 일정하게 측정되었다.



(그림 5) 재시작 시간에서 각 작업이 차지하는 비율

4. 관련 연구

도커 자체 만으로도 충분 할 수 있지만, IoT 환경에서 더 적합하도록 도커를 기반으로 한 프로젝트들이 있다. 오픈소스 프로젝트인 Eliot[6]은 도커와 쿠버네티스[7]의 기술과 개념을 재사용하여 IoT 장치의 한계에 초점을 맞추었다. 컨테이너를 실행할 수 있는 최소한의 요구사항으로 간단하게 컨테이너화 된 어플레이케이션을 관리할 수 있는 시스템이다. Raspberry Pi 의 공식 운영체제 중 하나인 Raspbian stretch lite 가 1.7GB 인 것에 비하여 EliotOS 는 70MB 보다 적은 용량을 차지한다.

도커가 만들고 관리하는 Moby 프로젝트를 기반으로 임베디드와 IoT 를 위한 컨테이너 엔진인 Balena[8]는 binary delta 업데이트라는 방법을 사용한다. 업데이트된 레이어를 그대로 가져오는 것이 아니라 바이너리에서 변경된 부분만을 가져옴으로써 기존의 Docker pull 방법보다 대역폭을 10~70 배 효율적으로 사용하며, 이미지의 크기는 3 배까지 줄일 수 있는 수단을 제공한다. 이러한 방법은 최악의 경우에도 Docker pull 과 같은 크기이다. 또한 업데이트시 메모리 사용율을 줄여 서비스 중인 다른 컨테이너에 영향을 최소화한다. 마찬가지로 Balena 또한 오픈소스 프로젝트이다.

5. 결론 및 향후연구

본 논문에서는 IoT 환경에서 도커 컨테이너의 서비스가 재시작 되는데 걸리는 시간이 시스템에 주어지는 부하에 의해 나타나는 성능 저하를 실험을 통해 확인하였다. 실험은 CPU, 메모리 입출력 그리고 디스크 입출력 작업에 의한 부하 유형에 대해 진행되었으며 디스크 입출력 작업에 의한 부하가 주어졌을 때 약 243.64%만큼 성능이 저하됨으로써 성능 저하에 가장 큰 영향을 주었다. 특히 재시작에 필요한 작업 중 create, connect 그리고 disconnect 작업이 많은 영향을 받게 된다는 것을 확인할 수 있었다.

하지만 부하가 주어지지 않았음에도 서비스를 재시작 하는데 10 초 정도 걸리는 것은 실제 비즈니스 환경에 치명적일 수 있다. 실험에서 일반적으로 컨테이너가 시작되는데 start 작업이 가장 큰 비율을 차지하는 것을 알 수 있다. 프로세스를 실행하는 코드를 읽기 위해 디스크 입출력 작업이 필요하기 때문이다. 따라서 컨테이너가 시작하는데 필요로 하는 디스크 입출력 작업량을 줄이는 방법에 관한 연구가 필요하다.

참고문헌

[1] Docker. https://www.docker.com/

[2] Paolo Bellavista, and Alessandro Zanni. Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi. In *International Conference on Distributed Computing and Networking* (ICDCN), 2017.

[3] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic Computing: A New Paradigm for Edge/Cloud Integration. In *IEEE Cloud Computing*, 2016.

[4] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems* (EuroSys), 2015.

[5] Stress-ng. http://kernel.ubuntu.com/~cking/stress-ng/

[6] Eliot. https://github.com/ernoaapa/eliot

[7] Kubernetes. https://kubernetes.io/

[8] Balena. https://github.com/resin-os/balena