Tepid Start: 유휴 Serverless 컨테이너의 관리 전략과 구현†,‡

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

Tepid Start: Managing Strategy for Idle Serverless Container and Implementation

Seunghyun Hwang*, Jihun Kang*, Kwangsik Chung**, Heonchang Yu*, Joonmin Gil***
*Dept. of Computer Science and Engineering, Korea University
**Dept. of Computer Science, Korea National Open University
***School of IT Engineering, Catholic University of Daegu

요 약

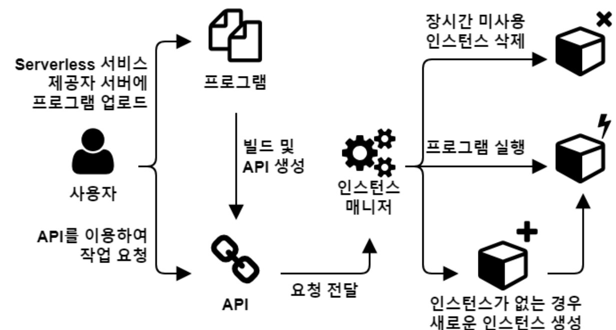
점점 더 많은 어플리케이션이 Serverless 컴퓨팅 기반으로 구현되고 있다. Serverless 서비스 제공자는 가용자원을 확보하기 위하여 요청이 장시간동안 발생하지 않은 서비스 인스턴스를 종료한다. 종료된 인스턴스에 대한 요청은 Cold Start 로 인한 지연시간이 발생하게 된다. 서비스 사용자는 Cold Start 를 방지하기 위해 인스턴스가 종료되지 않도록 주기적으로 의미 없는 요청을 하게 되고, 이는 Serverless 서비스 시스템에 불필요한 오버헤드를 발생시키게 된다. 따라서 이 논문은 이러한 불필요한 오버헤드를 줄이기 위한 전략을 제시하고 그 구현에 관해 설명한다.

1. 서론

최근 어플리케이션이 컨테이너와 마이크로 서비스 구조로 이동하면서 Serverless 컴퓨팅이 주목받게 되었다[1]. (그림 1)은 일반적인 Serverless 서비스의 구조에 대해 보여준다. Serverless 서비스의 사용자는 서비스 제공자의 서버에 실행될 프로그램과 리소스를 업로드하고 프로그램을 실행(Invoke)할 API 가 생성된다. 사용자는 실제로 서버를 구현하지 않았지만, API 를 이용하여 요청을 보내면 사용자의 프로그램이 실행되고 결과를 응답한다. 이처럼 Serverless 서비스의 사용자는 서버를 구축하기 위한 복잡한 구성과 유지관리를 할 필요가 없다. 게다가 Serverless 방식에서는 요청에 대한 요금만 부과되어(pay-for-use) 사용자는 서비스를 운영하는 비용도 절감할 수 있다.

Serverless 서비스 제공자는 인프라의 가용자원을 확보하기 위하여 요청이 장시간 동안 발생하지 않은 서비스 인스턴스를 종료한다. 따라서 사용자의 프로그램은 항상 즉시 실행가능한 Warm 상태가 아니다. 이러한 상태에서 요청이 발생하면 사용자 프로그램은 Cold Start 를 한다. 즉 사용자 프로그램이 실행될 새로운 컨테이너가 생성되고 프로그램이 메모리에 올라가는 시간이 필요하므로 실제 요청이 수행되기까지 지연시간이 발생하게 된다.

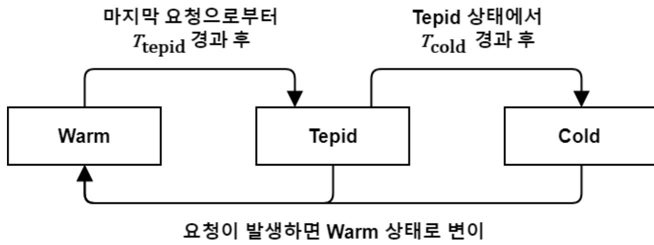
이 논문은 2016 년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. NRF-2016R1D1A3B03933370).



(그림 1) 일반적인 Serverless 서비스의 구조

Liang Wang *et.al.*[2]은 주요 Serverless 서비스에서 Cold Start 로 인한 지연시간을 측정했다. 결과에 따르면, 최대 16 초 정도의 서비스 지연이 발생했다. 이러한 지연을 회피하기 위해 서비스 사용자들은 자신의 프로그램을 실행하는 인스턴스가 종료되지 않도록 주기적으로 의미 없는 요청을 보낸다. 서비스 제공자로서는 장시간 요청이 없는 인스턴스를 지우지 못하게 되고 사실상 불필요한 모니터링을 수행하며, 해당 인스턴스는 요청을 수신하기 위한 상태에 있으므로 컴퓨팅 자원을 점유하게 된다.

따라서 본 논문에서는 컴퓨팅 자원의 낭비를 줄이기 위해 단시간 요청이 없는 인스턴스를 일시중지 시켰다가 요청이 발생하면 다시 시작하는 Tepid Start 전략을 제안하고 그 구현을 설명한다. 또한 다중 스레드 환경에서 구현의 안전성에 대해서 분석했다.



(그림 2) Tepid Start 전략에서 프로세스의 상태변화

이 논문의 구성은 다음과 같다. 2 장에서 Cold Start 문제에 대한 Serverless 제공자가 취할 수 있는 컨테이너 관리 전략을 제시하고, 그 구현에 관해 설명한다. 3 장에서는 제시한 전략에 대한 구현의 안전성을 설명하고, 4 장에서 성능을 분석하며 마지막으로 5 장에서는 결론과 본 논문에서 제시한 전략의 한계점을 서술한다.

2. 전략 및 구현

본 장에서는 제안하는 전략과 전략을 적용하기 위한 시스템의 구조, 그리고 구조에 따른 성능과 구현을 보여준다. 구조는 (그림 **1**)에서 설명한 요소상 구조는 동일하지만 인스턴스 매니저와 인스턴스 사이에 추가적인 통신이 필요하다. 구현에서 인스턴스는 다중 스레드 환경에서 동작하며 통신 채널은 FIFO 라고 가정한다.

2.1. 전략

본 논문이 제안하는 Tepid Start 전략은 (그림 2)와 같이 프로세스가 실행중인 상태인 Warm 상태와 프로세스가 종료된 상태인 Cold 상태 사이에 프로세스가 SIGSTOP Signal 로 인해 정지된 상태인 Tepid 상태를 추가하는 것이다. 정지된 프로세스는 스케줄러에서 고려를 하지 않기 때문에 CPU 자원을 사용하지 않게 된다. 하지만 프로세스가 종료된 것은 아니기 때문에 프로세스의 실행 정보는 메모리에 남아있게 된다. 정지된 프로세스는 SIGCONT Signal 을 받으면 실행이 재개된다. 기존에는 서비스가 마지막으로 호출된 시점에서 특정 Threshold 시간 T_{cold} 만큼 경과되면 Warm 상태에서 Cold 상태가 된다. 제안하는 전략에서는 시간 T_{tepid} 만큼 경과했을 때 Warm 상태에서 Tepid 상태로, 이후 시간 T_{cold} 만큼 경과하면 Tepid 상태에서 Cold 상태가 되는 것이다. Tepid 상태에서 서비스 요청이 발생하면 Warm 상태로 전이해 요청을 수행한다.

2.2. 구조

현재 Serverless 서비스 제공자들은 사용자 프로그램이 실행되는 시간에 제한을 두고 있다[3][4][5]. 이는 사용자의 프로그램이 예상치 못한 입 력에 적절한 처리를 하지 못해 종료되지 않고 장시간 실행되면서 발생하는 추가적인 과금으로 인한 사용자의 피해를 막기 위한 목적이다. 이러한 제한 시간을 T_{limit}이라고 했을 때, T_{tepid}는 T_{limit}보다 커야 한다. 사용자 프로그램이 정상적으로 실행되고 있는 중에 프로세스가 정지되면 안되기 때문이다. 하지만 T_{tepid} 는 작을수록 서비스 제공자 입장에선 좋지만, T_{limit}을 작게 한다는 것은 프로그램의 최대 실행 시간이 줄인다는 의미이다. 따라서 서비스 사용자 입자에서는 프로그램 실행시간에 대한 제약이 커지게 된다. 그렇지만 Cold Start 를 회피하기 위한 사용자의 의미 없는 요청은 짧은 시간에 끝나는데, 이를 고려하자면 T_{tepid}가 짧아야 하는 딜레마가 생긴다.

```
1.  T_tepid := N seconds
2.  runner := 0
3.
4.  function Suspend {
5.     request "suspending" to "Instance Manager";
6.  }
7.
8.  function Invoker {
9.     loop {
10.       listen request from user;
11.       async Invoke();
12.    }
13. }
14.
15. function Invoke {
16.    cancel Suspend();
17.    send ACK to "Instance Manager";
18.    increase runner by 1;
19.
20.    run user program;
21.
22.    lock(runner, called);
23.    decrease runner by 1;
24.    if called is 0 {
25.       Suspend() after T_tepid;
26.    }
27.    unlock(runner, called);
28. }
```

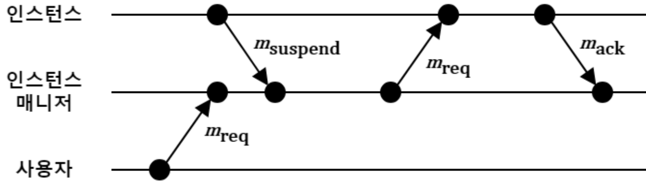
(그림 3) 인스턴스에 적용되는 알고리즘

본 논문이 제안하는 구현의 구조는 인스턴스 매니저가 인스턴스의 Tepid 상태 시기를 결정하는 것이 아니라, 인스턴스가 결정하도록 하는 것이다. 인스턴스는 사용자 프로그램을 실행시키는 주체이므로 사용자 프로그램의 실행상태를 알 수 있다. 따라서 실행이 종료된 시점이후 프로세스가 정지될 적절한 시기를 정할 수 있으므로 T_{tepid}를 T_{limit}보다 짧게 설정해서 사용자 프로그램이 실행되는 중간에 Tepid 상태가 되는 문제를 해결할 수 있다.

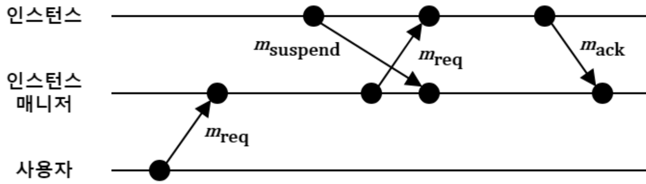
2.3. 구현

실제 구현에서 Serverless 인스턴스는 보통 컨테이너로 구현되거나[6][7][8] 컨테이너 환경이 제공된다[9][10]. 컨테이너 산업 표준을 정의하는 OCI(Open Container Initiative)의 컨테이너 Runtime 사양을 보면, kill 명령을 통해 컨테이너에서 실행되는 프로세스에 Signal 을 보낼 수 있다[11]. 따라서 SIGSTOP 과 SIGCONT Signal 을 보내 컨테이너의 프로세스를 정지하거나 다시 실행시킬 수 있으며, OCI 사양을 준수하는 컨테이너 Runtime 인 "runc"는 컨테이너를 정지하는 Pause 명령을 직접적으로 제공한다[12]. 따라서 본 논문의 구현은 Serverless 인스턴스가 컨테이너이며, Runtime 으로 "runc"를 사용한다고 가정한다. 즉 Serverless 인스턴스인 컨테이너에서 실행중인 모든 프로세스가 컨테이너 Runtime 의 명령을 이용하여 모두 중지되고 나중에 중지된 상태에서부터 실행이 재개될 수 있다.

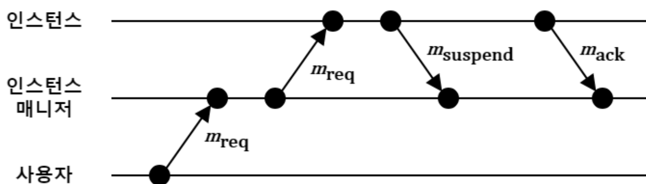
(그림 3)은 Serverless 인스턴스에 적용되는 알고리즘이며 메인 스레드는 *Invoker* 함수이다. *Invoker* 함수는 요청을 기다리는 10 번 줄에서 Blocked 상태이며 요청이 오면 *Invoke* 함수를 비동기 호출한 뒤, 다시 요청을 기다린다. *Invoke* 함수는 실제 사용자 프로그램을 실행하는 함수이다. 사용자 프로그램을 실행하기 전에, 인스턴스 매니저에 현재 인스턴스 프로세스 정지 요청을 하는 *Suspend* 함수의 예약된 호출



(a) 사용자 요청이 전달되기 전 중지 요청 도착



(b) 사용자 요청이 전달된 후 중지 요청 발생



(c) 사용자 요청이 전달된 후 중지 요청 도착

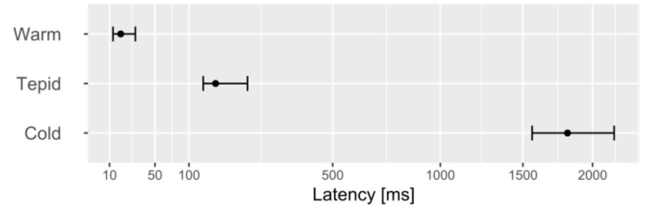
(그림 4). 인스턴스 매니저와 인스턴스의 통신 시나리오

을 취소하고 인스턴스 매니저에 ACK 를 보낸다. 그리고 현재 실행되고 있는 사용자 프로그램의 수를 나타내는 *runner* 변수를 1 증가시킨다. *Invoke* 함수는 사용자 요청에 의해 비동기로 호출되므로 다중으로 실행될 수 있다. 따라서 변수 *runner* 에 원자적 연산이 필요하다는 것에 유의해야한다. 사용자 프로그램이 종료되면 *runner* 변수를 1 감소시키고 변수가 0 이 된다면 T_{tepid} 경과 후에 *Suspend* 함수가 호출되도록 예약한다. 앞의 변수 runner 에 대한 원자적 연산과 마찬가지로 변수 runner 의 감소와 변수 called 의 참조는 임계영역에서 수행되어야 한다. 만약 T_{tepid}가 경과되기 전에 *Invoke* 함수가 호출되면 16 번 줄에서 예약된 *Suspend* 함수 호출을 취소하므로 *Suspend* 는 마지막 요청에 대해서만 호출된다. 따라서 사용자 프로그램이 실행되는 중간에 *Suspend* 가 호출되어 컨테이너가 정지되는 문제는 발생하지 않는다.

3. 안전성

본 논문의 구현은 다중 스레드 환경을 가정했으므로 (그림 3)의 *Suspend* 함수가 실행되어 정지가 요청되는 동시에 사용자 요청이 입력되어 *Invoke* 함수가 실행될 수 있다. 이는 사용자 프로그램이 실행되는 도중에 컨테이너가 정지될 수 있다는 것을 의미한다. 하지만 여전히 실제로 컨테이너를 정지하는 주체는 인스턴스 매니저이므로 사용자 요청의 유무에 따라 인스턴스 매니저가 정지 요청을 무시함으로써 문제가 해결될 수 있다.

(그림 4)는 앞에서 말한 시나리오가 발생할 수 있는 상황을 보여준다. m_{req} 는 사용자의 Serverless 호출 요청, $m_{suspend}$는 인스턴스의 정지 요청, 그리고 m_{ack}는 사용자의 Serverless가 호출되었다는 응답이다. (a), (b)의 경우 인스턴스가 마지막 요청 이후 T_{tepid} 가 경과하여 정지요청을 했지만



(그림 5) 상태에 따라 사용자 요청이 시작되는데 걸린 시간

인스턴스 매니저는 사용자 요청을 보내야 하므로 정지 요청을 무시할 수 있다. (c)의 경우 사용자 요청이 인스턴스에 전달된 후 인스턴스에서 중지 요청이 전달된 경우이다. 하지만 아직 사용자 요청에 대한 ACK 가 응답되지 않아 사용자 프로그램이 아직 실행되지 않았거나 실행 중인 것을 의미하므로 중지 요청을 무시할 수 있다.

만약 m_{req}가 $m_{suspend}$ 보다 먼저 도착하여 인스턴스 매니저가 컨테이너 중지 명령을 시작했더라도, 중지 명령을 취소하거나 즉시 컨테이너를 재개하면 되므로 사용자의 요청은 바로 시작될 수 있다. m_{req} 전송 후 m_{ack}가 응답되고, 곧이어 $m_{suspend}$가 오는 시나리오는 발생되지 않는다. 왜냐하면 인스턴스가 m_{ack}를 응답하기 전에 *Suspend* 호출을 취소했기 때문이다. 따라서 (그림 3) 알고리즘은 인스턴스 매니저가 사용자 요청의 유무에 따라 인스턴스 정지 요청을 무시함으로써 안전하다.

4. 성능

정지된 프로세스의 실행정보가 메모리에 남아있더라도, 프로세스를 재개하기 위한 명령이 "runc"를 포함한 여러 레이어를 거쳐 SIGCONT 시그널이 전송된다. 이러한 오버헤드가 얼마만큼의 영향을 끼치는 지 알아보기 위해 각 상태의 인스턴스가 요청된 시각으로부터 프로그램이 실행되기 직전 까지의 시간을 측정하는 실험을 진행하였다.

구현의 가정대로 Serverless 인스턴스는 컨테이너이며, 컨테이너 플랫폼은 Runtime 으로 "runc"를 사용하는 Docker 를 이용하였다. Warm Start 는 실행중인 컨테이너에 m_{req}를 전송하여 사용자 어플리케이션을 실행하는 것이다. Tepid Start 는 이미 존재하는 컨테이너가 "docker pause" 명령어로 인해 정지된 상태에서 "docker unpause" 명령어로 컨테이너를 재개시키는 것이고, Cold Start 는 컨테이너가 존재하지 않는 상태에서 "docker run" 명령어로 컨테이너를 실행시키는 것이다. "docker run" 명령어는 컨테이너를 실행하기 위해 필요한 이미지를 다운로드하고 컨테이너를 만드는 과정을 포함한다. 본 실험에서는 컨테이너를 만드는데 필요한 이미지가 이미 다운로드 되어 있는 상태에서 진행되었다. 모든 요청은 Serverless 인스턴스가 존재하는 물리 노드와 같은 물리 노드에서 이루어 졌다. Serverless 를 실행시키는 컨테이너 이미지는 Docker Hub 공식 이미지인 "node:8-alpine"을 기반으로 작성되었으며 각 상태에 따라 1,000 회씩 실험을 진행하였다. Docker Client 와 Server 버전은 18.09, Go version 은 1.10.4 이며 물리 노드의 사양 및 성능은 <표 1>과 같다.

<표 1> 실험 물리 노드 사양 및 성능

분류	사양 및 성능
OS	Ubuntu 18.04.1 LTS
Processor	Intel(R) Core(TM) i5-2500 3.30GHz
Mem Sequential Read	6477.32 MiB/sec
Disk Sequential Read	103.04 MiB/sec

(그림 5)는 실험 결과를 보여준다. 오차 측정값의 최소값과 최대값이며, 막대 위의 점은 중앙값을 나타낸다. Warm 상태에서 중앙값은 17ms, Tepid 상태에서 151ms, 그리고 Cold 상태에서 1,812ms 가 측정되었다. Warm 상태를 기준으로, Tepid Start 는 약 888%, Cold Start 는 약 10,658%의 오버헤드가 발생했다. Warm Start 와 Tepid Start 사이의 지연시간 차이와 Tepid Start 와 Cold Start 사이의 지연시간 차이의 비가 약 1,240%이지만, 888%의 오버헤드가 적은 것이 아니므로 너무 작은 T_{tepid} 설정은 서비스 품질을 떨어뜨릴 수 있다.

5. 결론

본 논문에서는 Serverless 서비스 사용자가 Cold Start 문제를 회피하기 위한 의미 없는 요청으로 인해 Serverless 인스턴스가 종료되지 못하고 무의미하게 컴퓨팅 자원을 점유하는 것을 완화하고자 Tepid Start 전략을 제시하고 그 구현과 구현이 안전하다는 것을 보였고, Tepid Start 로 인한 오버헤드를 측정하였다.

Tepid Start 전략은 인스턴스를 정지시키는 것으로 CPU 자원의 낭비는 막았지만 인스턴스가 메모리는 여전히 점유하고 있는 상태이다. 하지만 Serverless 서비스에 사용되는 컨테이너는 동일한 기반 이미지위에 작은 크기의 사용자 어플리케이션이 실행되므로 각 인스턴스가 차지하는 메모리의 크기가 크지 않다.

Docker 의 경우, 컨테이너가 정지되었더라도 해당 컨테이너의 가상 네트워크 보조 프로세스는 여전히 실행되어 CPU 자원을 차지하고 있다. 이는 컨테이너 Runtime 을 관리하는 Daemon 에서 관리되는 자원이기 때문에 본 연구에서 다루지 않았다. Tepid Start 전략을 도입할 경우 컨테이너 보조 프로세스도 함께 정지하는 것도 고려될 수 있다. 특히 네트워크 자원을 정지하게 되면 인스턴스가 재개 요청을 수신하지 못할 수도 있으므로 Oakes *et.al.*[13]이 제안한 SOCK 컨테이너와 같이 외부와 통신하기 위해 UNIX 소켓을 사용할 수도 있다.

참고문헌

[1] Baldini, Ioana, et al. "Serverless computing: Current trends and open problems." Research Advances in Cloud Computing. Springer, Singapore, 2017. 1-20.

[2] Wang, Liang, et al. "Peeking behind the curtains of serverless platforms." 2018 {USENIX} Annual Technical Conference (USENIX ATC 18). 2018.

[3] AWS Lambda Limits. [online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/limits.html

[4] Azure Functions scale and hosting. 2018. [online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale

[5] Google Cloud Functions Execution Environment. 2018. [online]. Available: https://cloud.google.com/functions/docs/concepts/exec

[6] Apache OpenWhisk Documentation. [online]. Available: https://openwhisk.apache.org/documentation.htm

[7] OpenFaaS Introduction. [online]. Available: https://docs.openfaas.com/

[8] Understanding Container Reuse in AWS Labmda. 2014. [online]. Available: https://aws.amazon.com/ko/blogs/compute/container-reuse-in-lambda/

[9] Serverless in Azure. [online]. Availabe: https://azure.microsoft.com/en-us/solutions/serverless/

[10] Cloud Functions serverless platform is generally available. 2018. [online]. Available: https://cloud.google.com/blog/products/gcp/cloud-functions-serverless-platform-is-generally-available

[11] Open Container Initiative runtime-spec. 2017. [online]. Available: https://github.com/opencontainers/runtime-spec/blob/master/runtime.md#kill

[12] Open Container Initiative runc. 2018. [online]. Available: https://github.com/opencontainers/runc/blob/master/man/runc-pause.8.md

[13] Oakes, Edward, et al. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers." *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018.