J. Internet Comput. Serv.
ISSN 1598-0170 (Print) / ISSN 2287-1136 (Online)
http://www.jics.or.kr
Copyright ⓒ 2024 KSII

GraalVM 및 Virtual Thread 환경에서 API Gateway의 성능 평가

Performance evaluation of API Gateway in GraalVM and Virtual Thread environment

조 동 일[1]*
Dong-il Cho

요 약

API 게이트웨이는 클라우드 외부의 API 클라이언트가 클라우드 내부 서비스와 통신할 수 있는 단일 진입점을 제공하는 고가용성 구성요소로서 병목 현상을 발생시킬 위험이 크고 서비스 변경 시 재배포가 필요하다. 여러 API 게이트웨이가 구현되고 있는 Java 언어는 배포와 운영 성능의 문제를 극복하기 위해 GraalVM Native Image와 Virtual Thread라는 기술을 발표하였다. Java 애플리케이션에 이들 기술을 적용하기 위해서는 소스 코드 및 배포 절차를 변경해야 한다. 본 연구에서는 API 게이트웨이가 GraalVM Native Image와 Java Virtual Machine(JVM) 기반으로 동작했을 때와 Virtual Thread와 Reactive 스레드 처리 방식 간의 성능을 측정하고 분석하였다. 본 연구에서는 배포 성능과 운영 성능의 평가를 위해 평가지표를 선정하였고 네 가지 환경에서 평가지표의 성능을 측정하고 평가하였다.

☞ 주제어 : API Gateway, GraalVM, Virtual Thread, Reactive, Microservice Architecture, Cloud Computing

ABSTRACT

An API gateway is a high-availability component that provides a single entry point for API clients outside the cloud to connect with services inside the cloud. It has a high risk of creating bottlenecks and requires redeployment when services change. The Java language, in which several API gateways are implemented, announced technologies called GraalVM Native Image and Virtual Thread to overcome problems with deployment and operational performance. Applying these technologies to Java applications requires changes to the source code and deployment procedures. In this study, the performance of the API gateway was measured and analyzed when it operated based on GraalVM Native Image and Java Virtual Machine(JVM) and between Virtual Thread and Reactive thread processing methods. In this study, evaluation indicators were selected to evaluate deployment performance and operational performance, and the performance of the evaluation indicators was measured and evaluated in four environments.

☞ keyword : API Gateway, GraalVM, Virtual Thread, Reactive, Microservice Architecture, Cloud Computing

1. 서 론

Microservice Architecture(MSA)의 필수 요소로 클라우드 외부의 API 클라이언트가 클라우드 내부 서비스와 통신할 수 있는 단일 진입점을 제공하는 API 게이트웨이는 고가용성 구성요소로서 병목 현상을 발생시킬 위험이 크고 서비스의 API를 노출하려면 API 게이트웨이를 변경해야 한다[1,2,3]. 이런 특징으로 인해 API 게이트웨이는 동적 확장 및 배포가 가능해야 하며 운영 중 안정적인 성능이 보장되어야 한다.

Java는 널리 사용되는 서비스 개발 언어로 Spring Native, Quarkus, Micronaut, Helidon 등 다양한 MSA 개발 프레임워크가 공급되고 있고 Java 언어로 개발된 다양한 API 게이트웨이 구현체가 배포되고 있다. Java 서비스 응용프로그램은 Java Virtual Machine(JVM)을 함께 배포해야 하는 제약으로 인해 배포 사이즈가 크고 CPU와 메모리 사용량이 운영체제에서 직접 실행되는 프로그래밍 언어에 비해 높다는 문제점이 있다[4,5]. 이 문제들을 해결하기 위해 GraalVM Native Image와 Virtual Thread라는 기술을 발표되었다. 이 기술들은 Java 언어로 개발된 애플리케이션에 적용이 가능하지만 기존 구현체와 배타적인

1 Division of It Software, Shingu College., Seongnam, 13174, Korea
* Corresponding author (chodongil@yahoo.co.kr)
[Received 29 March 2024, Reviewed 27 May 2024(R2 09 July 2024), Accepted 10 July 2024]

http://dx.doi.org/10.7472/jksii.2024.25.4.39

여러 가지 제약사항으로 인해 기존 응용프로그램에 적용하기 위해서는 소스 코드와 배포 시나리오의 변경이 필요할 수 있다[6,7].

본 연구에서는 API 게이트웨이에 GraalVM Native Image와 Virtual Thread 그리고 대안 기술이었던 JVM과 Reactive를 각각 적용하여 배포 성능과 운영 성능을 측정하고 평가하였다. 본 연구의 결과는 API 게이트웨이의 주요 성능 지표에 따라 각 기술의 적용을 결정하는 근거 자료로 활용될 수 있다.

2. 관련 연구

2.1 API Gateway

MSA는 애플리케이션을 각각 자체 프로세스에서 실행되는 경량 메커니즘과 통신하는 독립적인 서비스 집합으로 개발하는 방법이다[2]. 클라이언트는 이론적으로 각 MSA에 직접 요청을 보낼 수 있으나 이런 접속 방식은 클라이언트가 접속해야 하는 모든 MSA의 주소를 미리 알고 있어야 하기 때문에 구현과 배포가 어렵고 MSA를 캡슐화할 수 없게 한다[1].

API 게이트웨이는 모든 클라이언트에 대해 MSA 서비스에 접속할 수 있는 단일 진입점을 제공하고 요청 라우팅, API 조합, 프로토콜 변환 그리고 인증 및 권한 처리와 같은 엣지 기능을 제공할 수 있다. API 클라이언트는 API 게이트웨이를 통해 호출 횟수를 줄이고 클라이언트 특화 프로토콜을 유지하면서 MSA API와 통신할 수 있다[1].

API 게이트웨이는 모든 클라이언트의 단일 진입점으로 병목 현상을 발생시킬 수 있는 고가용성 구성요소이고 서비스 API를 노출하기 위해 API 게이트웨이를 변경해야 한다[3]. 특히 Java 언어로 구현된 API 게이트웨이는 JVM과 함께 배포되어 배포 사이즈가 크고 운영 시 높은 CPU 및 메모리가 필요하며 서비스 시작 시간이 느린 문제가 있다[8].

2.2 Reactor와 Spring WebFlux

Servlet API가 처음 출시되었을 때 구현 컨테이너의 대부분은 요청당 전용 스레드를 사용했다. 전용 스레드는 요청 처리가 완료되고 응답이 클라이언트에 전송될 때까지 스레드가 차단된다. 현재는 서버에 연결되는 HTTP 요청 수가 크게 늘어나 많은 웹 애플리케이션들은 스레드를 차단하는 것이 더 이상 가능하지 않다.

이 문제를 개선하기 위해 Reactive 프로그래밍 분야가 발전하게 되었고 Spring WebFlux를 사용하여 Reactive 웹 애플리케이션의 개발이 가능해졌다. 반응성을 갖추기 위해 Spring은 Reactive Streams API의 구현으로 Project Reactor를 활용한다[9]. 이벤트 기반 비동기 프레임워크인 Spring WebFlux는 역압이 가능하고 스레드를 차단하지 않는 클라이언트 및 서버를 제공할 수 있다[4].

Reactive 프로그래밍은 요청 처리 로직을 람다 표현으로 작성된 작은 단계로 분해한 다음 API를 사용하여 순차적인 파이프라인으로 구성한다. 파이프라인의 각 단계는 서로 다른 스레드에서 실행될 수 있으며 모든 스레드는 서로 다른 요청에 속하는 각 단계를 끼워 넣는 방식으로 실행한다. 이러한 Reactive 프로그래밍은 애플리케이션의 동시성 단위인 비동기 파이프라인이 플랫폼의 동시성 단위가 아니기 때문에 Java 플랫폼과 상충 되어 소스 코드가 복잡하기 때문에 개발과 디버깅이 어렵다[7].

2.3 Virtual Thread

애플리케이션이 플랫폼과 조화를 유지하면서 확장할 수 있도록 하려면 요청 별 스레드 스타일을 유지해야 하고 더 풍부한 스레드를 공급해야 한다. Java 언어와 JVM이 스레드 스택을 다른 방식으로 사용하기 때문에 운영체제 스레드를 더 효율적으로 구현하는데 한계가 있다. 그러나 JVM이 스레드 스택을 구현할 때 필요에 따라 운영체제 스레드와 연결하는 방식으로 구현하는 것은 가능하며 이 접근 방식을 사용하면 단일 JVM에서 수백만 개의 Virtual Thread를 사용할 수 있다[10].

요청 별 스레드 스타일의 애플리케이션 코드는 요청 동안 Virtual Thread에서 실행될 수 있지만 Virtual Thread는 CPU에서 계산을 수행하는 동안에만 운영체제 스레드를 소비한다. 그 결과 투명하게 달성되는 것을 제외하고 비동기 스타일과 유사한 확장성을 제공할 수 있다. Virtual Thread에서 실행되는 코드가 Java에서 차단 I/O 작업을 호출하면 JVM은 비차단 운영체제 호출을 수행하고 나중에 다시 시작할 수 있을 때까지 Virtual Thread를 자동으로 일시 중단한다[11]. Java 개발자에게 Virtual Thread는 단순히 생성하기에 저렴하고 거의 무한히 많은 스레드이다. 하드웨어 활용도는 최적에 가까워 높은 수준의 동시성과 결과적으로 높은 처리량을 허용하는 반면 애플리케이션은 Java 플랫폼의 다중 스레드 설계 및 툴링과 조화를 유지한다[7]. Virtual Thread는 JDK19에서 소개되었으며 JDK21에서 정식 제공되었다[7].

2.4 GraalVM Native Image

GraalVM은 Graal 컴파일러, GraalVM Native Image, Truffle Language 구현 프레임워크 및 Sulong(LLVM)을 비롯한 여러 중요한 하위 시스템을 통합하여 광범위한 생태계를 제공하는 차세대 고성능 다중 언어 가상머신을 구축하기 위해 시작된 OpenJDK의 Graal 프로젝트의 일부로 개발되었다[12].

GraalVM Native Image는 Java 코드를 Substrate VM 런타임 시스템에서 필요한 런타임 구성요소와 함께 독립 실행형 파일로 컴파일한다. 결과적으로 생성된 애플리케이션은 JVM보다 시작 시간이 빠르고 실행 중 메모리 오버헤드가 적다[4].

Native Image 생성에 사용되는 정적 Ahead-of-Time(AOT) 컴파일러는 Graal 컴파일러와 같지는 않지만 대응되도록 조정되었다[12]. Graal 컴파일러에는 모듈식 아키텍처와 추론적 컴파일러 최적화가 있으며 이를 위해서 추가적인 연속 역최적화를 실행해야 한다[13]. 컴파일러는 코드 성능을 최적화하기 위해 지점 간 분석 결과를 조정하고 이미지 힙의 일부를 적재하여 힙이 미리 채워진 실행 가능한 이미지를 생성한다. 결과적으로 GraalVM Native Image는 완전한 플랫폼별 실행 파일로 Native Image를 실행하기 위해 JVM을 제공할 필요가 없다[6,14].

JVM용 애플리케이션과 달리 GraalVM Native Image 애플리케이션은 실행 파일을 생성하는 과정에 기본 진입점에서 애플리케이션 코드를 정적으로 분석하는 단계가 포함되는데 이는 Java의 동적 클래스 로딩 구현과 상충된다[8]. 따라서 API 게이트웨이와 포함하는 라이브러리의 구현 방식에 따라 적용 가능 여부가 결정되며 GraalVM Native Image를 적용하기 위해서는 라이브러리를 변경하거나 대체 구현이 필요할 수 있다[6].

3. 성능 평가

3.1 평가 요소

운영 성능 관점에서 API 게이트웨이는 서비스 제공 중 안정적으로 CPU와 메모리를 사용해야 하며 높은 처리량을 제공해야 한다.

서버 애플리케이션의 확장성은 대기시간, 동시성 및 처리량과 관련된 Little의 법칙으로 관리된다. 주어진 요청 처리 시간에 대해 애플리케이션이 동시에 처리하는 요청 수는 처리량(L)에 비례하며 평균 체류시간(W)과 평균 요청 도달 수(λ)를 이용하여 (1)로 나타낼 수 있다[15].

$$L = \lambda W \tag{1}$$

각 요청을 처리하는 동안 스레드를 점유하면 애플리케이션은 처리량이 증가함에 따라 스레드 수를 증가시켜야 한다[7]. JVM의 스레드 처리 방식은 고전적인 전용 스레드 방식에서 성능 향상을 위해 Reactive와 Virtual Thread 방식이 제시되었으며 본 연구에서는 이들 각각의 운영 성능 비교를 위해 CPU와 메모리 사용량 그리고 Throughput을 측정 지표로 선정하였다.

API 게이트웨이는 서비스 클라이언트의 요청 증가에 따라 동적으로 확장되고, 서비스가 변경되면 새롭게 패키징(P) 된 후 배포(D)되고 실행(S)된다. 이 과정은 (2)와 같이 표현할 수 있다.

$$Distribution\ Time \fallingdotseq P() + D(F) + S() \tag{2}$$

(2)에서 각 과정은 환경적인 영향을 받기 때문에 함수로 표현되었다. 이 식에서 배포(D)는 배포 환경 이외에 배포할 파일의 사이즈가 영향을 줄 수 있기 때문에 인수로 추가 되었다. 따라서 본 연구에서 배포 성능의 측정 지표는 패키징 시간(Packaging Time), 배포할 파일의 크기(File Size) 그리고 서비스 시작 시간(Start Time)으로 선정하였다.

3.2 평가 방법

평가대상은 스레드 처리 관점에서 Virtual Thread와 Reactive 그리고 실행 환경 관점에서 Native Image와 JVM 환경이다. 이 환경을 조합하여 측정한 평가대상은 다음과 같다.

- Virtual Thread(*vt*)를 사용하는 API 게이트웨이의 JVM 동작 환경(*jvm*)
- Virtual Thread(*vt*)를 사용하는 API 게이트웨이의 Native Image 동작 환경(*ntv*)
- Reactive(*rt*) 방식으로 스레드를 사용하는 API 게이트웨이의 JVM 동작 환경(*jvm*)
- Reactive(*rt*) 방식으로 스레드를 사용하는 API 게이트웨이의 Native Image 동작 환경(*ntv*)

본 연구에서는 성능 비교를 위해 이전 연구에서 구현한 Stream-based API Gateway(SAG)를 활용하였다[16].

Virtual Thread와 Reactive는 구현 방식에 차이가 있다. SAG는 Spring WebFlux를 기반으로 Reactive 형태로 스레드를 처리하도록 구현되었으므로 기존 SAG와 같은 기능을 제공하면서 Virtual Thread로 동작할 수 있는 버전을 추가 구현하였다. Virtual Thread는 Reactor에 비해 직관적인 구현이 가능하였고 디버깅이 편리하였다.

이 두 가지 버전의 API 게이트웨이를 각각 JVM과 Native Image로 패키징하여 측정하고자 하는 각각의 평가지표 별 수치를 측정하였다.

API 게이트웨이의 컴파일과 실행을 위해 JDK는 Virtual Thread와 GraalVM Native Image를 모두 지원하는 Liberica NIK 23.1.2.r21-nik를 사용했으며 적용된 프레임워크와 버전은 표 1과 같다.

(표 1) 구현 환경
(Table 1) Implementation Environment

Reactive API Gateway	Virtual Thread API Gateway
spring-boot 3.2.2 spring-webflux 6.1.3	spring-boot 3.2.2 spring-webmvc 6.1.3

배포 및 운영 성능은 다음과 같은 하드웨어 환경에서 측정되었다.

- CPU: Intel i5-1235U(4.4 GHz, 10 core 12 Threads)
- Memory: 16 GiB
- SSD: PCIe M.2 2280 SSD 500GB
- OS: Ubuntu Linux 22 LTS

3.2.1 배포 성능 측정

식별된 배포 성능 지표는 각각 다음과 같은 방법으로 측정하였다.

- Packaging Time: API 게이트웨이의 소스 코드를 컴파일하고 패키징하는데 소요되는 시간을 5회 측정한 후 평균값을 계산하였다. 패키징 도구는 Maven 3.9.6을 사용하였으며 패키징에 사용한 Maven 명령은 다음과 같다.

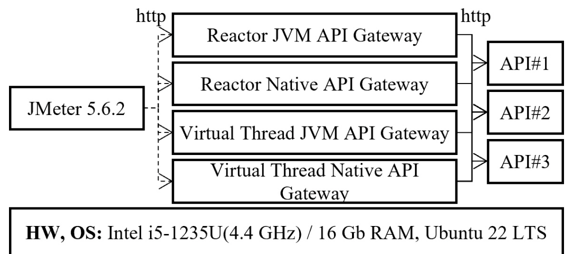
JVM: mvn clean package
Native Image: mvn -Pnative clean native:compile

- File Size: 패키징한 API 게이트웨이를 실행하기 위해 필요한 파일들의 크기를 측정하였다. Native Image로 배포된 경우 측정 대상은 API 게이트웨이 실행 파일이고 JVM에서 구동하는 API 게이트웨이는 API 게이트웨이 jar 파일과 JVM의 크기를 합한 값을 측정하였다. JVM의 크기는 JVM에 포함된 모든 파일의 크기를 합한 값을 적용하였다.

- Start Time: 실행 가능 상태로 패키징 된 API 게이트웨이를 실행한 직후 시간과 서비스 시작이 완료 로그가 출력된 시간 간의 차이를 5회 측정하여 평균값을 산출하였다.

3.2.2 운영 성능 측정

본 연구에서는 API 게이트웨이의 운영 성능을 측정하기 위해 그림 1과 같이 JMeter를 이용해 서비스 클라이언트의 요청을 가상화하여 API 게이트웨이에 전송하고 요청이 처리되는 동안에 각각의 운영 성능 평가지표 수치를 측정하였다.



(그림 1) 운영 성능 측정 환경
(Figure 1) Operational Performance Measurement Environment

위 환경에서 세 가지 REST API를 서비스하는 웹 앱과 테스트 대상이 되는 네 가지 API 게이트웨이를 설치한 후 테스트 대상이 되는 API 게이트웨이를 각각 실행하여 테스트하였다.

테스트 시나리오는 다음과 같다.

1. JMeter가 API 게이트웨이에 요청 정보를 전송한다.
2. API 게이트웨이는 요청에 대응되는 3개의 REST API를 동시에 호출한다.
3. 3개의 REST API는 각각 16KB, 5KB, 7KB의 JSON 응답 데이터를 API 게이트웨이에 반환한다.
4. API 게이트웨이는 세 API 호출 결과를 Stream 기반으로 조합하여 JMeter에 전송한다[16].

JVM 기반의 응용프로그램은 높은 메모리 공간과 긴 준비시간으로 인해 높은 시작 비용이 발생하지만 동시 가비지 수집과 최대 처리량 조건에서 복원력을 높이는 기능으로 점차 안정화된다[8]. 이 특징으로 인해 충분히 긴 시간으로 테스트할 경우 일정 시간 이후 JVM 응용프로그램의 자원 소모는 안정화되어 실제 서비스 상황에서의 자원 소모를 측정할 수 있다. 따라서 우리는 요청 횟수에 제한을 두지 않고 충분히 긴 시간 동안 지속적으로 부하를 주입하여 자원의 사용량을 측정하였다.

사전 모의 테스트 결과 JVM 기반의 API 게이트웨이는 약 60초 이후 안정된 성능을 보였고 실제 테스트에서 우리는 JMeter를 5분 동안 위 시나리오를 반복적으로 실행하였다. JMeter의 부하 설정 수치는 다음과 같다.

- Number of Threads(users) : 100
- Ramp-up period(seconds) : 5
- Duration(seconds) : 300

평가지표 별 측정 방법은 다음과 같다.

- CPU 사용량: JMeter가 API 게이트웨이에 부하를 전달하는 동안 API 게이트웨이 프로세스의 CPU 사용량을 다음 명령을 이용하여 1초 단위로 측정하였다.

pidstat -u -p %PID% 1

- 메모리 사용량: JMeter가 API 게이트웨이에 부하를 전달하는 동안 API 게이트웨이 프로세스의 메모리 사용량을 다음 명령을 이용하여 1초 단위로 측정하였다.

pidstat -r -p %PID% 1

- Throughput: JMeter는 기능적 동작 성능을 적재하고 분석하는 역할을 하며 서버, 네트워크 또는 기타 항목에 대한 상당한 로드를 시뮬레이션하여 정적 및 동적 자원의 사용량을 측정하고 복원력을 평가할 수 있다[17]. JMeter는 테스트 중 요청 당 응답의 처리량을 초당 처리량으로 환산하여 제공하는데 이 값을 처리량 결과로 활용하였다.

3.3 평가 결과 및 분석

배포 성능 측정 결과는 표 2와 같이 Reactive와 Virtual Thread 간에는 큰 차이를 보이지는 않았으나 JVM과 Native는 큰 차이를 보였다.

〈표 2〉 배포 성능 측정 결과
〈Table 2〉 Deployment Performance Measurement Results

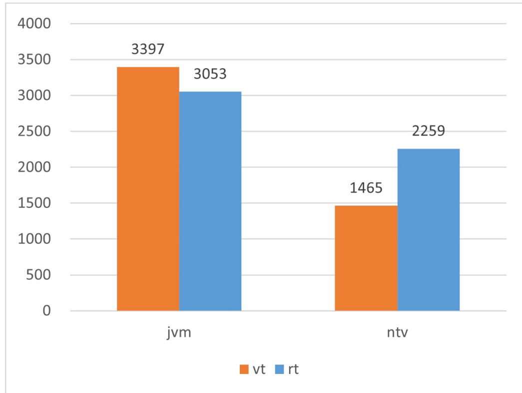
	Reactive		Virtual Thread	
	JVM	Native	JVM	Native
Packaging Time(s, □)	2.950 (±0.2)	127.2 (±3.4)	2.361 (±0.3)	123.4 (±1.7)
Start Time(ms, □)	1133.8 (±36.5)	27.6 (±0.5)	1171.6 (±27.5)	38.8 (±2.2)
File Size(MB)	637	91	636	90

Packaging Time에서 JVM은 3초 이내의 빠른 시간에 패키징이 가능했지만, Native Image는 약 125초로 JVM 대비 42배의 시간을 기록했다. GaalVM이 Native Image를 생성하기 위해 사용하는 AOT 컴파일러는 컴파일 시간 동안 주석 메타데이터를 미리 계산하기 위해 모든 종속성 및 구성 주입을 계산하여 실행 시 모든 메타데이터 작업을 제거한다. 즉 프레임워크 인프라에 해당하는 모든 것은 실행 시 더 높은 메모리와 시작 시간을 피하기 위해 컴파일 시점에 수행된다[18].

반면 JVM에서 실행되는 Just-in-Time(JIT) 컴파일러는 Java 코드를 바이트코드로 변환하며 Java Hotspot VM이 실행되는 시점에 기계어 코드로 변환한다. JIT는 프로그램을 실행하는 동안 기계어 코드를 생성하여 컴파일 시간이 짧은데, 반해 AOT 컴파일러는 프로그램 실행 전 기계어 코드를 생성하며 Native API 게이트웨이는 120초 이상 더 긴 Packaging Time이 필요하였다.

Start Time과 File Size에서는 Native Image가 유리한 것으로 나타났다. Native Image는 API 게이트웨이가 시작할 때 필요한 설정 프로세스가 컴파일 시점에 미리 정의되어 빠른 시작 시간을 기록하였으나 JVM은 Java Hotspot VM이 바이트코드를 기계어로 번역하는 과정과 주석 메타데이터를 해석하여 실행하는 과정이 포함되어 약 35배 긴 시작 시간을 기록하였다. File Size에서 JVM은 JDK가 가지는 기본 크기인 601MB를 추가 배포해야 하기 때문에 Native Image에 비해 약 7배 크기로 측정되었다.

운영 성능 평가지표에서 처리량은 그림 2와 같이 JVM이 Native Image에 비해 Virtual Thread는 56%, Reactive는 26% 높게 측정되었다. JVM은 동시 가비지 수집기를 사용하고 최대 처리량 조건에서 복원력을 높이는 기능 덕분에 대기시간이 점차 줄어든다. 반면 Native Image는 시작과 동시에 최대 성능을 제공하고 균일하게 유지된다[14,19]. 그 결과 [19]에서 밝힌 바와 같이 JVM은 Native Image에 비해 최대 처리량이 더 높게 나타났다.



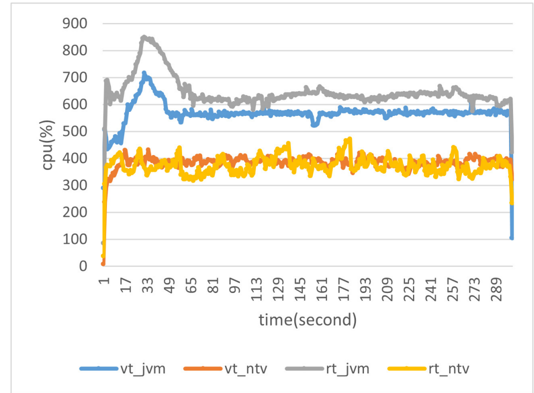
(그림 2) 초당 요청 처리량
(Figure 2) Request Throughput per Second

Reactive는 JVM과 Native Image 간의 처리량 차이가 26%로 Virtual Thread에 비해 상대적으로 작게 나타났다. SAG는 Reactive를 위해 Spring Webflux와 reactor-netty를 사용하는데 reactor-netty는 비동기식 설계를 지원하기 위해 Netty 프레임워크 기반으로 로컬 TCP/HTTP/UDP 클라이언트 및 서버를 포함하여 비차단 및 역압 지원 네트워크 런타임을 제공한다[20]. 또한 reactor-netty는 Virtual Thread와 유사한 이벤트 루프를 사용하여 I/O를 처리하도록 설계되어 있으며 Native I/O 버퍼를 Java 버퍼로 복제하는 과정을 생략할 수 있어 JVM에서 실행될 때에도 상당 부분 Native Image와 같은 방식으로 동작한다[21]. 특히 API 게이트웨이의 경우 주요 처리 패턴이 메모리 복제이므로 그 효과가 크게 나타났다.

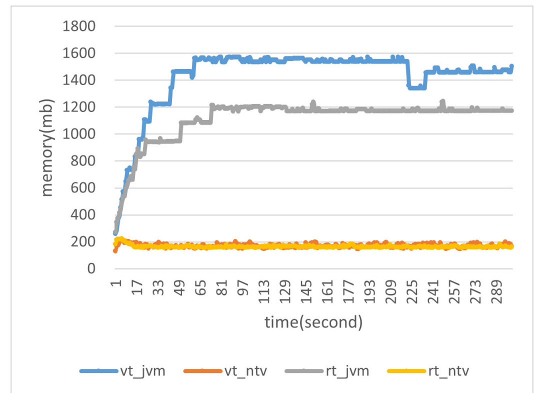
CPU와 메모리 사용량은 그림 3과 그림 4와 같이 Native Image로 실행했을 때 Reactive와 Virtual Thread는 실행 직후부터 종료될 때까지 균일한 수치를 보였는데, 반해 JVM으로 실행한 API 게이트웨이는 실행 시점부터 점차 증가하다가 테스트 65초를 지난 시점부터 안정화되었다. 안정화 구간인 65 ~ 290초 구간의 CPU 사용량의 평균값을 비교한 결과 Virtual Thread일 때 Native Image는 약 387%로 567%를 사용한 JVM에 비해 약 68%를 사용하였으며 Reactive는 Native Image일 때 374%로 628%를 사용한 JVM에 비해 약 60%의 CPU 사용량을 보였다.

메모리 사용량에서 Virtual Thread는 Native Image일 때 168MB를 사용하여 1,515MB를 사용한 JVM에 비해 약 11%를 사용률을 보였고, Reactive는 Native Image일 때 162MB를 사용하여 1,176MB를 사용한 JVM의 약 14%를 사용하였다.

즉 Native Image는 JVM 대비 약 60% 대의 CPU와 약 10%대의 메모리 자원만으로 Virtual Thread는 42%, Reactor는 73%의 처리량을 처리할 수 있었다.



(그림 3) CPU 사용량 변동
(Figure 3) CPU Usage Fluctuating



(그림 4) 메모리 사용량 변동
(Figure 4) Memory Usage Fluctuating

4. 결 론

API 게이트웨이는 클라우드 외부의 API 클라이언트가 클라우드 내부 서비스와 통신할 수 있는 단일 진입점을 제공하는 고가용성 구성요소로서 병목 현상을 발생시킬 위험이 크고 신속한 동적 확장과 재배포가 필요하다.

Java는 널리 사용되는 서비스 개발 언어로 다양한 MSA 개발 프레임워크와 API 게이트웨이 구현체가 개발되고 있는 개발 언어인데, JVM을 함께 배포해야 하고 CPU와 메모리 사용량이 Native 애플리케이션에 비해 높

다는 문제점이 있다. 이런 문제를 극복하고자 최근 GraalVM Native Image와 Virtual Thread라는 기술이 발표되었고, Virtual Thread는 JDK21에 공식 배포되었다. 이들은 Java로 개발된 애플리케이션의 성능 향상을 목표로 하지만 이미 이런 문제를 해결하기 위한 대안 기술과 배타적인 제약사항으로 인해 기존 기술이 적용된 애플리케이션에 적용하기 위해서는 소스 코드 및 배포 시나리오의 변경이 필요하다.

본 연구에서는 API 게이트웨이를 대상으로 GraalVM Native Image와 JVM 그리고 Virtual Thread와 Reactive를 적용했을 때 성능의 차이를 측정하고 비교하였다. 성능 측정을 위해 우리의 이전 연구에서 개발한 Reactive 기반의 SAG를 Virtual Thread 버전으로 추가 구현하였고 JVM과 Native Image로 패키징 한 후 각각에 대해 배포 성능과 운영 성능을 측정하고 상호 비교 평가하였다.

실험 결과 배포 성능에서 Reactive와 Virtual Thread 간에는 큰 차이가 없었으나 JVM과 Native Image 간에는 Packaging Time은 Native Image가 JVM의 약 42배의 시간이 소요되었고, 시작 시간은 JVM이 약 35배 느렸으며 File Size는 JVM이 약 7배 컸다.

운영 성능 관점에서 처리량은 JVM에서 동작했을 때 Virtual Thread와 Reactive는 각각 56%와 26% 높은 처리량을 보였다. 반면 Native Image는 시작과 동시에 안정적인 응답 성능을 제공하였다. JVM의 워밍업이 끝난 안정화 구간에서 Virtual Thread는 Native Image로 동작했을 때 JVM에 비해 약 68%의 CPU 사용량과 약 11%의 메모리 사용량을 보였으며, Reactive는 Native Image로 동작했을 때 JVM에 비해 약 60%의 CPU 사용량과 약 14%의 메모리 사용량을 보였다.

본 연구의 실험 결과 API 게이트웨이의 수정이 빈번하고 잦은 배포가 필요한 경우에는 패키징 시간이 짧고 개발과 디버깅이 편리한 JVM 기반의 Virtual Thread를 선택할 수 있다. 반면 수정은 거의 일어나지 않으나 스케일인/아웃이 발생할 수 있고 안정적인 클라우드 자원 소모가 필요한 경우에는 Native Image 기반의 Reactive를 적용하는 것이 유리하다고 볼 수 있다.

이와 같이 본 연구의 결과는 API 게이트웨이와 유사한 서비스 애플리케이션이 가져야 하는 성능 기준의 중요도에 따라 적합한 기술의 적용을 결정하는 근거 자료로 활용될 수 있다.

참고문헌(Reference)

[1] C. Richardson, “Microservices Patterns,” Manning, pp.152-160, 253-291, 2018.

[2] Q. Xiong and W. Li, “Design and Implementation of Microservices Gateway Based on Spring Cloud Zuul,” in Proc. of 3rd International Conference on Computer Information and Big Data Applications, pp.1-5, 2022. https://ieeexplore.ieee.org/document/9899125

[3] X. Gao, R. Liu and X. Lin, “API Gateway Optimization Architecture Based on Heterogeneous Hardware Acceleration,” in Proc. of IEEE 3rd International Conference on Information Technology and Big Data and Artificial Intelligence, Vol.3, pp.863-868, 2023. https://doi.org/10.1109/ICIBA56860.2023.10165387

[4] A. Sharma, K. Tahiliani and G. P. Dubey, “Reactive-Optimized Sentence Detection In Kubernetes Using OpenNLP And Native GraalVM Image With Framework Metric Comparison,” in Proc. of 2023 4th International Conference for Emerging Technology, pp.1-9, 2023. https://doi.org/10.1109/INCET57972.2023.10170347

[5] H. D. Long, T. Vergilio and A. Kor, “Comparative Performance and Energy Efficiency Analysis of Jvm Variants and Graalvm in Java Applications,” SSRN, 2023. http://dx.doi.org/10.2139/ssrn.4373169

[6] GraalVM Native Image Support. Available online: https://docs.spring.io/spring-boot/docs/current/reference/html/native-image.html

[7] R. Pressler and A. Bateman, “JEP 444: Virtual Threads,” 2023. Available online: https://openjdk.org/jeps/444

[8] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, “Initialize once, start fast: application initialization at build time,” Proceedings of the ACM on Programming Languages, Vol. 3, Article 184, pp.1-29, 2019. https://doi.org/10.1145/3360610

[9] M. Deinum, D. Rubio, J. Long, “Spring 6 Recipes A Problem-Solution Approach to Spring Framework fifth Edition,” pp.205-240, Apress, 2023.

[10] B. Evans, “Going inside Java’s Project Loom and virtual threads,” Java magazine, 2021. Available online: https://blogs.oracle.com/javamagazine/post/going-inside-javas-project-loom-and-virtual-threads

[11] D. Beronić, P. Pufek, B. Mihaljević and A. Radovan, “On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM,” in Proc. of 2021 44th International Convention on Information, Communication and Electronic Technology, pp.1684-1689, 2021. https://doi.org/10.23919/MIPRO52101.2021.9596855

[12] M. Šipek, B. Mihaljević, and A. Radovan, “Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM,” in Proc. of 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, pp.1671-1676, 2019. https://doi.org/10.23919/MIPRO.2019.8756917

[13] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer and H. Mössenböck, “Graal IR: An Extensible Declarative Intermediate Representation,” in Proc. of the Asia-Pacific Programming Languages and Compilers Workshop, pp.1-9, 2013. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6688214dab5456c75c99f8171846242e09d4f5e3

[14] Why GraalVM?. Available online: https://www.graalvm.org/why-graalvm/

[15] J. D. C. Little, “Little’s Law as Viewed on Its 50th Anniversary,” Operations Research, Vol.59, No.3, pp.536-549, 2011. https://doi.org/10.1287/opre.1110.0940

[16] D. I. Cho, “Stream-based API composition for stable API Gateway,” Journal of Internet Computing and Services, Vol.25, No.1, pp.1-8, 2024. http://dx.doi.org/10.7472/jksii.2024.25.1.1

[17] A. Ismail, A. Y. Ananta, S. N. Arief and E. N. Hamdana, “Performance Testing Sistem Ujian Online Menggunakan Jmeter Pada Lingkungan Virtual,” Jurnal Informatika Polinema, Vol.9. No.2, pp.159-164, 2023. https://doi.org/10.33795/jip.v9i2.1190

[18] M. Šipek, D. Muharemagić, B. Mihaljević and A. Radovan, “Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus,” in Proc. of 2020 43rd International Convention on Information, Communication and Electronic Technology, pp.1746-1751, 2020. https://doi.org/10.23919/MIPRO48935.2020.9245290

[19] O. Šelajev, “Pedal to the metal: High-performance Java with GraalVM Native Image,”, Java Magazine, 2021. Available online: https://blogs.oracle.com/javamagazine/post/pedal-to-the-metal-high-performance-java-with-graalvm-native-image

[20] Project Reactor. Available online: https://projectreactor.io/

[21] Netty Project. Available online: https://netty.io/

◐ 저 자 소 개 ◑

조 동 일(Dong-il Cho)
2003년 수원대학교 기계공학과(공학사)
2008년 숭실대학교 정보과학 대학원 소프트웨어 공학과(공학석사)
2012년 숭실대학교 대학원 컴퓨터학과(공학박사)
2003년~2024년 ㈜ 토마토시스템 기술연구소 연구소장
2024년~현재 신구대학교 IT소프트웨어과 조교수
관심분야 : Microservice Architecture, 웹 애플리케이션 아키텍처, WEB UI/UX, etc.
E-mail : chodongil@yahoo.co.kr