Smart Media Journal / Vol.15, No.2 / ISSN:2287-1322
https://dx.doi.org/10.30693/SMJ.2026.15.2.142

Gunicorn Gevent Worker 모델 기반 Flask SaaS 웹 서비스 고동시성 및 성능 최적화 연구

(Performance Optimization Study of Flask SaaS Web Service Based on Gunicorn Gevent Worker Model for High Concurrency)

이해인*

(Hae In Lee)

요 약

본 연구는 Flask 기반 SaaS 웹 서비스의 동시 접속 성능 최적화 방안을 제시한다. 기본 Sync Worker 모델의 Blocking I/O 문제를 해결하기 위해 Gunicorn을 Gevent Worker로 전환하고 Monkey Patching을 적용하였다. 실험 결과, Sync Worker 방식(workers=10)은 10명 초과 접속 시 응답 지연이 발생한 반면, Gevent Worker(workers=8, worker_connections=300)는 100명 동시 접속 환경에서도 안정적 응답을 유지하였다. 특히 50명 동시 접속 환경에서 78.1%의 응답 시간 개선을 달성하였다. 워커 수를 감소시키면서도 worker_connections를 통해 더 많은 동시 연결을 처리할 수 있어, 메모리 효율성과 고동시성을 동시에 달성하였다. 본 연구는 제한된 자원으로 고동시성을 달성하는 비용 효율적 최적화 방안을 제시한다.

■ 중심어 : Flask ; Gunicorn ; Gevent ; 고동시성 ; 비동기 ; I/O ; 성능 최적화 ; SaaS

Abstract

This study presents a performance optimization method for Flask-based SaaS web services under high concurrency. To address blocking I/O issues in the default Sync Worker model, we converted Gunicorn to Gevent Worker and applied Monkey Patching. Experimental results showed that while Sync Worker (workers=10) experienced response delays under multiple concurrent connections, Gevent Worker (workers=8, worker_connections=300) maintained stable response times even with 100 concurrent connections. Notably, we achieved a 78.1% response time improvement in a 50-user concurrent environment. By reducing the number of workers while increasing worker_connections, we achieved stable handling of more simultaneous connections, thereby accomplishing both memory efficiency and high concurrency. This study demonstrates a cost-effective optimization approach for achieving high concurrency with limited resources.

■ **keywords :** Flask ; Gunicorn ; Gevent ; High Concurrency ; Asynchronous I/O ; Performance Optimization ; SaaS

Ⅰ. 서 론

1. 연구 배경

. 클라우드 기반 SaaS(Software as a Service) 시장은 지속적으로 성장하고 있으며, 웹 애플리케이션의 확장성과 성능이 서비스 품질을 결정하는 핵심 요소가 되었다. Python Flask 프레임워크는 마이크로 웹 프

* 정회원, 공주대학교 컴퓨터교육과
이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2000-0000000).

접수일자 : 2025년 12월 27일
게재확정일 : 2026년 01월 06일
교신저자 : 이해인 e-mail : hilove4361@naver.com

레임워크로서 간결한 구조와 높은 개발 생산성으로 인해 스타트업과 중소규모 서비스에서 널리 채택되고 있다. 특히 AI 서비스, 교육 플랫폼, 데이터 분석 도구 등 다양한 분야에서 Flask를 기반으로 한 웹 서비스가 구축되고 있다.

2. 문제 제기

Flask는 WSGI(Web Server Gateway Interface) 표준을 따르며, 일반적으로 Gunicorn과 같은 WSGI 서버와 함께 사용된다. 그러나 Gunicorn의 기본 Sync Worker 모델은 동기(Synchronous) 방식으로 동작하며, 각 워커 프로세스가 한 번에 하나의 요청만 처리할 수 있는 구조적 제약이 있다[1]. 이러한 구조에서는 데이터베이스 쿼리, 외부 API 호출 등 I/O 작업 중 Blocking이 발생하면 해당 워커가 응답을 반환할 때까지 다른 요청을 처리할 수 없어 전체 시스템의 처리량이 저하된다. 특히 동시 접속자 수가 10명을 초과하면 요청이 대기열(Backlog)에 쌓이거나 타임아웃이 발생하여 사용자 경험이 크게 저하되는 문제가 발생한다.

3. 연구 목적 및 범위

본 연구는 AWS EC2 환경의 Docker 컨테이너에서 운영되는 Flask 웹 서비스를 대상으로, Gunicorn Worker 모델을 Sync에서 Gevent로 변경함으로써 고동시성 환경에서의 성능을 최적화하는 방안을 제시한다. 구체적으로 Gevent의 Greenlet 기반 코루틴과 Monkey Patching을 활용하여 비동기 I/O를 구현하고[2], 동시 접속자 수 증가에 따른 응답 시간, 처리량, 시스템 자원 사용률을 측정하여 성능 개선 효과를 정량적으로 분석한다. 이를 통해 제한된 서버 자원으로 더 많은 사용자를 수용할 수 있는 비용 효율적인 최적화 방안을 제시하고자 한다.

II. 선행 연구

1. Python WSGI와 Gunicorn 구조

WSGI(Web Server Gateway Interface)는 Python 웹 애플리케이션과 웹 서버 간의 표준 인터페이스로, PEP 3333에 정의되어 있다. WSGI 서버는 HTTP 요청을 받아 Python 애플리케이션으로 전달하고, 애플리케이션의 응답을 HTTP 형식으로 변환하여 클라이언트에 반환하는 역할을 수행한다[3].

Gunicorn(Green Unicorn)은 Unix 계열 시스템에서 널리 사용되는 WSGI HTTP 서버로, Prefork 모델을 기반으로 한다. 이 모델에서는 마스터 프로세스가 여러 워커 프로세스를 생성하고 관리하며, 마스터 프로세스는 워커의 상태를 모니터링하고 필요 시 재시작한다. 각 워커 프로세스는 독립적으로 요청을 처리하며, 이러한 프로세스 격리 구조는 안정성을 제공한다.

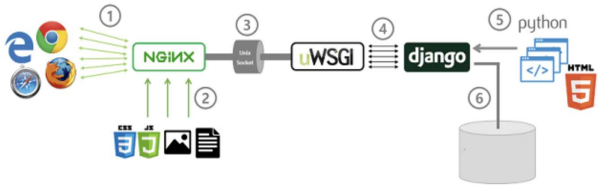

그림 1. WSGI 아키텍처 및 Gunicorn 구조

그림 1은 WSGI 기반 웹 애플리케이션의 전체 아키텍처를 보여준다. 클라이언트의 요청은 먼저 Nginx 리버스 프록시를 거쳐 Unix Socket을 통해 WSGI 서버로 전달된다. Nginx는 정적 파일(CSS, JavaScript, 이미지 등)을 직접 처리하여 애플리케이션 서버의 부하를 줄이고, 동적 요청만 WSGI 서버로 전달한다. WSGI 서버인 Gunicorn은 마스터 프로세스와 다수의 워커 프로세스로 구성되며, 각 워커는 독립적으로 Flask 애플리케이션을 실행하여 비즈니스 로직을 처리한다. 최종적으로 데이터베이스와의 통신을 통

해 데이터를 조회하거나 저장한다. 이러한 계층 구조는 각 컴포넌트의 역할을 명확히 분리하여 유지보수성과 확장성을 향상시킨다.

Gunicorn의 마스터 프로세스는 워커의 생명주기를 관리하며, 워커가 비정상 종료되거나 메모리 누수로 인한 문제가 발생하면 자동으로 재시작한다. 또한 graceful restart를 지원하여 서비스 중단 없이 설정 변경이나 코드 배포를 수행할 수 있다. 이러한 아키텍처는 프로덕션 환경에서 안정적인 서비스 운영을 가능하게 한다.

2. Sync Worker와 Gevent Worker 비교

Gunicorn은 다양한 Worker 클래스를 지원하며, 각각 다른 동시성 모델을 구현한다. Sync Worker는 가장 기본적인 모델로, 각 워커 프로세스가 순차적으로 요청을 처리한다. 이 방식은 구현이 단순하고 CPU 집약적 작업에 적합하지만, I/O 대기 시간 동안 워커가 유휴 상태로 남아 있어 자원 활용률이 낮다는 단점이 있다.

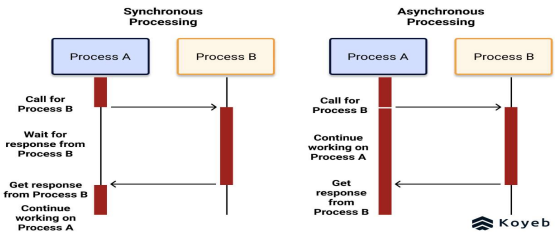

그림 2. Synchronous와 Asynchronous 처리 방식 비교

그림 2는 동기(Synchronous)와 비동기(Asynchronous) 처리 방식의 근본적인 차이를 보여준다. 좌측의 Sync Worker 모델에서는 Process A가 외부 프로세스(예: API 호출, 데이터베이스 쿼리)를 호출하면 응답을 받을 때까지 완전히 블로킹되어 다른 작업을 수행할 수 없다. 빨간색으로 표시된 대기 구간 동안 워커는 CPU와 메모리를 점유하고 있지만 실질적인 작업을 수행하지 못하는 비효율이 발생한다. 반면 우측의 Gevent Worker 모델에서는 Process A가 I/O 작업을 시작한 후 응답을 기다리는 동안 다른 작업을 계속 수행할 수 있어 자원 활용률이 크게 향상된다.

Sync Worker의 경우, 워커 수를 늘리면 더 많은 동시 요청을 처리할 수 있지만, 각 워커는 독립적인 프로세스이므로 메모리 오버헤드가 크다. 일반적으로 2GB RAM 환경에서 Flask 애플리케이션의 경우 워커당 약 100-150MB의 메모리를 소비하므로, 실질적으로 10~15개 정도의 워커만 실행할 수 있다. 또한 프로세스 간 컨텍스트 스위칭 비용이 발생하여 동시 접속자 수가 증가할수록 성능이 급격히 저하된다.

반면 Gevent Worker는 Greenlet 라이브러리를 기반으로 하는 코루틴 기반 동시성 모델을 제공한다. Greenlet은 Python의 경량 코루틴 구현으로, 운영체제 스레드보다 훨씬 적은 메모리를 사용하며 빠른 컨텍스트 스위칭이 가능하다. Gevent는 libev 이벤트 루프를 사용하여 Non-blocking I/O를 구현하고, Monkey Patching 기법을 통해 표준 라이브러리의 블로킹 함수들을 비동기 버전으로 대체한다. 이를 통해 하나의 워커 프로세스가 수백 개의 동시 연결을 효율적으로 처리할 수 있다.

Gevent Worker의 worker_connections 파라미터는 각 워커가 동시에 처리할 수 있는 최대 연결 수를 설정한다. 예를 들어 workers=8, worker_connections=300으로 설정하면 이론적으로 최대 2,400개의 동시 연결을 처리할 수 있다. 실제로는 메모리, CPU, 네트워크 대역폭 등의 제약이 있지만, Sync Worker에 비해 월등히 많은 동시 연결을 지원한다.

3. Gevent의 Greenlet 동작 메커니즘

Gevent의 핵심은 Greenlet 기반의 협력적 멀티태스킹이다. Greenlet은 Python C 확장 모듈로 구현된 경량 코루틴으로, 모든 Greenlet은 단일

OS 프로세스 내에서 실행되며 협력적으로 스케줄링된다. 한 번에 하나의 Greenlet만 실행되며, 명시적으로 제어권을 양보(yield)할 때만 다른 Greenlet으로 전환된다.

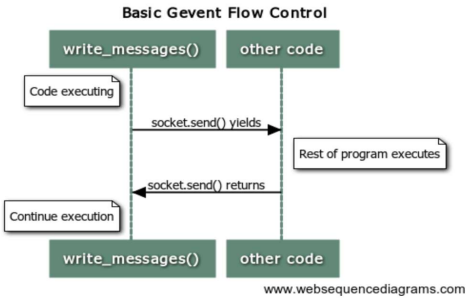

그림 3. Gevent의 Greenlet 동작 메커니즘

그림 3은 Gevent의 Greenlet이 협력적 멀티태스킹을 수행하는 과정을 보여준다.

write_messages() 함수가 socket.send()를 호출하면, Monkey Patching에 의해 해당 함수는 자동으로 yield하여 제어권을 이벤트 루프에 반환한다. 이때 다른 Greenlet(other code)이 실행되며, socket.send() 작업이 완료되면 다시 write_messages()로 돌아와 실행을 계속한다. 이러한 메커니즘을 통해 하나의 워커 프로세스 내에서 여러 I/O 작업을 동시에 처리할 수 있다.

Gevent는 libev 또는 libuv 이벤트 루프 위에 구축되어 있으며, 이를 통해 Non-blocking I/O를 제공한다. 이벤트 루프는 파일 디스크립터, 타이머, 시그널 등 다양한 이벤트를 모니터링하며, 이벤트가 발생하면 해당 Greenlet을 실행 가능 상태로 변경한다. 이러한 방식으로 수백 개의 Greenlet이 효율적으로 관리된다.

Monkey Patching은 Gevent의 핵심 기능으로, 런타임에 Python 표준 라이브러리의 블로킹 함수들을 비동기 버전으로 교체한다. gevent.monkey.patch_all() 함수를 호출하면 다음 표 1과 같은 모듈들이 자동으로 패치된다.

표 1. Monkey Patching을 통한 표준 라이브러리 모듈 변환

모듈	Monkey Patching 변환 결과
socket	모든 소켓 I/O가 비동기로 전환
ssl	SSL/TLS 연결이 비동기로 처리
threading	Thread 객체가 Greenlet으로 대체
select	I/O 멀티플렉싱이 이벤트 루프 기반으로 변경
subprocess	서브프로세스 실행이 비동기로 처리

이를 통해 requests, urllib3, psycopg2, pymysql 등 대부분의 I/O 라이브러리가 코드 수정 없이 비동기로 동작한다. 예를 들어, requests.get()을 호출하면 Monkey Patching에 의해 내부적으로 사용되는 socket이 비동기 소켓으로 대체되어, HTTP 요청 중 자동으로 다른 Greenlet으로 전환된다.

다만 Gevent Worker에도 한계가 존재한다. CPU 집약적 작업의 경우 Global Interpreter Lock(GIL)로 인해 병렬 처리가 불가능하며, 하나의 Greenlet이 CPU를 독점하면 다른 Greenlet이 실행되지 못한다. 또한 일부 C 확장 모듈(예: numpy, pandas의 특정 함수)은 Monkey Patching과 호환되지 않을 수 있다. 따라서 CPU 집약적 작업이나 호환되지 않는 라이브러리를 사용하는 경우, Celery와 같은 분산 작업 큐로 분리하여 처리하는 것이 권장된다.

4. Gevent의 Greenlet 동작 메커니즘

Docker 컨테이너 환경에서는 cgroup(Control Groups)을 통한 자원 제한이 중요하다. 메모리, CPU, 네트워크 대역폭 등의 자원을 컨테이너별로 할당하고 격리함으로써 다중 서비스 환경에서의 안정성을 확보할 수 있다. cgroup은 Linux 커널 기능으로, 프로세스 그룹에 대한 자원 사용을 제한하고 우선순위를 관리하며 사용량을 모니터링한다. Docker는 이를 활용하여 각 컨테이너가 할당된 자원 범위 내에서만 동작하도록 보장한다.

그림 4. Docker 컨테이너 기반 웹 애플리케이션 배포 구조

그림 4는 본 연구에서 구현한 Docker 컨테이너 기반 웹 애플리케이션 배포 구조를 보여준다. 다수의 클라이언트(Chrome 브라우저)로부터 병렬로 요청(Petición)이 발생하면, Docker 컨테이너 내부의 Nginx가 80번 포트에서 이를 수신한다. Nginx는 리버스 프록시로서 Unix Socket을 통해 8080번 포트의 Gunicorn으로 요청을 전달한다. Gunicorn은 그림 1에서 설명한 WSGI 서버로서, 다수의 워커 프로세스를 관리하며 각 워커는 Python Flask 애플리케이션을 실행한다. 처리 결과는 동일한 경로를 역순으로 거쳐 클라이언트에게 응답(Respuesta)된다.

이러한 구조는 Ubuntu 운영체제 위에서 단일 Docker 컨테이너로 패키징되어 실행되며, 컨테이너 격리를 통해 독립적인 실행 환경을 보장한다. Docker 컨테이너를 사용함으로써 개발 환경과 프로덕션 환경의 일관성을 확보하고, AWS EC2와 같은 클라우드 환경에서의 배포와 확장이 용이해진다. 또한 필요 시 동일한 컨테이너 이미지를 여러 인스턴스로 복제하여 수평 확장(Horizontal Scaling)을 수행할 수 있다.

Docker Compose를 사용하면 다중 컨테이너 애플리케이션을 YAML 파일로 정의하고 관리할 수 있다. 예를 들어, Nginx 컨테이너, Gunicorn/Flask 컨테이너, MySQL 컨테이너, Redis 컨테이너를 각각 독립적으로 실행하면서도 동일한 네트워크에서 통신하도록 구성할 수 있다. 이러한 마이크로서비스 아키텍처는 각 서비스의 독립적인 배포, 확장, 롤백을 가능하게 하여 운영 유연성을 크게 향상시킨다.

AWS EC2에서는 인스턴스 타입에 따라 vCPU 수와 메모리 용량이 결정되며, 이에 맞춰 Gunicorn 워커 수를 설정해야 한다. 일반적으로 권장되는 워커 수는 (2 × CPU + 1) 공식을 따른다[2]. 이는 Sync Worker를 기준으로 한 권장사항으로, CPU 코어당 2개의 워커를 할당하고 1개의 추가 워커를 두어 I/O 대기 중에도 요청을 처리할 수 있도록 한다. 예를 들어 4 vCPU 환경에서는 workers=9를 설정하는 것이 일반적이다.

그러나 I/O 집약적 애플리케이션에서는 이보다 많은 워커를 사용하거나 비동기 워커 모델을 채택하는 것이 더 효율적이다. Gevent Worker를 사용할 경우, 워커 수는 상대적으로 적게 설정하되 worker_connections를 높게 설정하는 것이 효과적이다. 예를 들어 4 vCPU 환경에서 Gevent Worker는 workers=8-10, worker_connections=200-300으로 설정하여 총 1,600-3,000개의 동시 연결을 처리할 수 있다. worker_connections 값은 메모리 용량과 예상 동시 접속자 수에 따라 조정하며, 각 연결당 약 2-5MB의 메모리를 소비한다고 가정하여 계산한다.

Docker 컨테이너의 자원 제한은 docker run 명령어의 옵션 또는 Docker Compose 파일의 resources 섹션을 통해 설정한다. 주요 설정 항목은 다음 표 2와 같다.

표 2. Docker 컨테이너 자원 제한 설정

자원 항목	설정 방법 및 설명
메모리 제한	--memory 옵션으로 컨테이너가 사용할 수 있는 최대 메모리를 제한함. 예를 들어 --memory=2g는 2GB로 제한한다. 메모리 제한을 초과하면 컨테이너 내부 프로세스가 OOM(Out Of Memory) Killer에 의해 종료될 수 있으므로, 애플리케이션의 예상 메모리 사용량에 20-30% 여유를 두고 설함.
CPU 제한	--cpus 옵션으로 컨테이너가 사용할 수 있는 CPU 코어 수를 제한함. 예를 들어--cpus=2.0은 2개 코어로 제한함. 또한--cpu-shares를 통해 여러 컨테이너 간 CPU 시간 배분 비율을 설정함.
네트워크 대역폭	tc(Traffic Control) 명령어나 Docker 플러그인을 통해 네트워크 대역폭을 제한 가능하며, 특정 컨테이너의 과도한 네트워크 사용이 다른 서비스에 영향을 미치는 것을 방지함.

Docker Compose를 사용한 오케스트레이션 환경에서는 Gunicorn 설정을 환경 변수로 외부화

하여 배포 환경에 따라 유연하게 조정할 수 있다. 예를 들어, docker-compose.yml 파일에서 WORKERS, WORKER_CLASS, WORKER_CONNECTIONS 등을 환경 변수로 정의하고, .env 파일이나 CI/CD 파이프라인에서 이를 주입한다. 이를 통해 개발 환경에서는 적은 워커로, 프로덕션 환경에서는 최적화된 워커 설정으로 동일한 이미지를 사용할 수 있다.

Health check는 Docker 컨테이너의 상태를 주기적으로 모니터링하는 중요한 기능이다. Dockerfile의 HEALTHCHECK 명령어나 Docker Compose의 healthcheck 섹션을 통해 설정하며, 일반적으로 HTTP 엔드포인트(예: /health)에 요청을 보내 응답 상태를 확인한다. Health check가 연속으로 실패하면 Docker는 해당 컨테이너를 unhealthy 상태로 표시하고, orchestrator(Docker Swarm, Kubernetes 등)는 자동으로 컨테이너를 재시작하거나 트래픽을 다른 정상 컨테이너로 우회시킨다.

AWS ECS(Elastic Container Service)나 Kubernetes와 같은 컨테이너 오케스트레이션 플랫폼을 사용하면 더욱 고급 기능을 활용할 수 있다. Auto Scaling을 통해 CPU 사용률이나 요청 수에 따라 컨테이너 인스턴스를 자동으로 증감시킬 수 있으며, Blue-Green 배포나 Canary 배포를 통해 무중단 배포가 가능하다. 또한 Service Mesh(예: Istio)를 도입하면 컨테이너 간 통신의 모니터링, 트래픽 제어, 보안 강화를 선언적으로 관리할 수 있다.

본 연구에서는 AWS EC2 t3.medium 인스턴스(2 vCPU, 4GB RAM)에서 Docker Compose를 사용하여 Nginx, Gunicorn/Flask, MySQL, Redis 컨테이너를 구성하였다. Gunicorn 컨테이너에는 메모리 2GB, CPU 1.5 코어를 할당하였으며, Gevent Worker를 사용하여 workers=8, worker_connections=300으로 설정하였다. 이러한 구성을 통해 제한된 자원으로도 높은 동시성을 달성할 수 있었다.

III. 시스템 설계 및 구현

1. 입력 구조 및 의미 기반 해석

본 연구에서 구현한 시스템은 Client – Nginx – Gunicorn – Flask App – Database의 다층 구조로 설계되었다. Nginx는 리버스 프록시로서 정적 파일 서빙과 로드 밸런싱을 담당하고, Gunicorn은 WSGI 서버로서 Flask 애플리케이션을 실행하며, Flask는 비즈니스 로직을 처리하고 MySQL 데이터베이스와 통신한다. 전체 시스템은 Docker Compose를 통해 오케스트레이션되며, AWS EC2 인스턴스에 배포된다. 시스템 구성 요소는 다음 표 3과 같다.

표 3. 시스템 구성 요소

구성 요소	역할
Nginx 1.21	리버스 프록시 및 로드 밸런서
Gunicorn 20.1	WSGI HTTP 서버
Flask 2.3	Python 웹 프레임워크
MySQL 8.0	관계형 데이터베이스
Redis 7.0	캐싱 및 세션 스토어
Celery 5.3	비동기 작업 처리

2. 초기 설정 및 문제점 (Case A)

초기의 시스템은 Gunicorn의 기본 Sync Worker 모델을 사용하였으며, workers=10으로 설정하여 10개의 워커 프로세스를 운영하였다.

표 4. Case A 소스코드 일부

gunicorn_config.py (Case A)
bind = '0.0.0.0:5000' workers = 10 worker_class = 'sync' timeout = 30 keepalive = 2

이 설정에서는 10개의 Sync Worker가 생성되며, 각 워커는 한 번에 하나의 요청만 처리할 수 있다. 실제 운영 중 다음과 같은 문제점이 발견되었다.

1) 동시 접속자 10명 초과 시 응답 시간이 급격히 증가
2) 워커 소진으로 인한 502 Bad Gateway 에러 발생
3) 외부 API 호출 시 전체 시스템 응답 지연
4) 데이터베이스 쿼리 대기 중 워커 블로킹

로그 분석 결과, I/O 작업 중 워커가 블로킹되어 다른 요청을 처리하지 못하는 것으로 확인되었다. 특히 Gemini API, OpenAI API 등 외부 AI 서비스 호출 시 평균 3-5초의 응답 시간이 소요되어, 10개 워커로는 동시에 10명의 사용자만 처리할 수 있었다.

3. 최적화 구현 (Case B)

문제 해결을 위해 Gunicorn Worker 모델을 Gevent로 변경하고 Monkey Patching을 적용하였다. 핵심 전략은 워커 수를 10개에서 8개로 감소시켜 메모리 오버헤드를 줄이면서도, worker_connections=300으로 설정하여 각 워커가 처리 가능한 동시 연결 수를 대폭 증가시키는 것이었다.

표 5. Case B 소스코드 일부

gunicorn_config.py (Case B)

```
from gevent import monkey
monkey.patch_all()

bind = '0.0.0.0:5000'
workers = 8
worker_class = 'gevent'
worker_connections = 300
timeout = 120
keepalive = 5
```

또한, 주요 최적화 적용 사항은 다음 표 6과 같다.

표 6. 주요 최적화 적용 사항

설정 항목	최적화 내용
Monkey Patching	monkey.patch_all()을 통해 socket, ssl, threading 등 표준 라이브러리의 블로킹 함수를 비동기 버전으로 교체
Worker 수 최적화	10개에서 8개로 감소시켜 메모리 오버헤드를 약 20% 줄이면서도, worker_connections=300으로 설정하여 각 워커당 처리 가능한 동시 연결 수를 대폭 증가시켜 총 2,400개(8*300)의 동시 연결 처리 가능. 이를 통해 메모리 효율성과 동시성을 동시에 달성
Worker Connections	각 워커가 최대 300개의 동시 연결 처리 가능 (Sync Worker는 워커당 1개만 처리)
Timeout 연장	AI API 호출 등 장시간 작업을 위해 30초에서 120초로 증가

Monkey Patching은 런타임에 표준 라이브러리의 모듈을 비동기 버전으로 교체하는 기법이다. 예를 들어, socket.socket() 함수를 gevent의 비동기 소켓으로 대체함으로써, 기존 코드 수정 없이 비동기 I/O를 구현할 수 있다. 이는 requests, urllib, 데이터베이스 드라이버 등 대부분의 I/O 관련 라이브러리에 자동으로 적용된다.

Ⅳ. 성능 실험 및 분석

1. 실험 환경

성능 측정은 AWS EC2 t3.medium 인스턴스(2 vCPU, 4GB RAM)에서 수행되었다. 부하 테스트 도구로는 Locust를 사용하였으며, 동시 접속자(Virtual Users)를 5명에서 100명까지 점진적으로 증가시키면서 응답 시간과 처리량을 측정하였다. 테스트 시나리오는 실제 서비스 사용 패턴을 반영하여 다음과 같이 구성하였고 실험 환경은 표 7과 같다.

1) 사용자 로그인 (MySQL 쿼리)
2) AI 학습 계획 생성 (Gemini API 호출, 평균 3초)
3) 학습 이력 조회 (Redis 캐시 + MySQL 쿼리)
4) 음성 대화 처리 (Google TTS/STT API 호출)

표 7. 실험 환경

항목	사양
서버	AWS EC2 t3.medium (2vCPU, 4GB RAM)
운영체제	Ubuntu 22.04 LTS
컨테이너	Docker 24., Docker Compose 2.20
부하 테스트 도구	Locust 2.15

2. 실험 결과

Case A(Sync Worker, workers=10)와 Case B(Gevent Worker, workers=8, worker_connections=300)의 성능을 비교한 결과는 다음 표 8과 같다.

표 8. 동시 접속자 수에 따른 평균 응답 시간 비교

동시 접속사	Case A 응답시간 (초)	Case A 응답시간 (초)	개선율 (%)
5	3.2	3.1	3.1
10	4.1	3.2	22.0
30	15.8	4.8	69.6
50	28.3	6.2	78.1
100	Timeout	9.5	–

실험 결과, Case A(Sync Worker, workers=10)는 동시 접속자가 10명을 초과하면서부터 응답 시간이 급격히 증가하였다. 30명에서는 평균 15.8초, 50명에서는 28.3초가 소요되었으며, 100명에서는 대부분의 요청이 타임아웃되었다. 반면 Case B(Gevent Worker, workers=8, worker_connections=300)는 100명의 동시 접속자 환경에서도 평균 9.5초의 안정적인 응답 시간을 유지하였다. 특히 50명 동시 접속 환경에서 78.1%의 응답 시간 개선을 달성하였다.

3. 고찰

성능 개선의 핵심 요인은 다음과 같이 분석된다. 첫째, Monkey Patching을 통한 비동기 I/O 구현으로 워커가 I/O 대기 중에도 다른 요청을 처리할 수 있게 되었다. 특히 외부 AI API 호출 시 3-5초의 대기 시간 동안 수백 개의 다른 요청을 처리할 수 있어 전체 처리량이 크게 증가하였다. 둘째, worker_connections=300 설정으로 각 워커가 최대 300개의 동시 연결을 유지할 수 있어, 8개 워커로 총 2,400개의 동시 연결을 처리할 수 있게 되었다. 셋째, 워커 수를 10개에서 8개로 감소시킴으로써 메모리 사용량을 약 20% 절감하면서도 동시 처리 능력은 오히려 240배 증가(10개 → 2,400개)하는 효과를 얻었다.

Ⅴ. 결론

본 연구에서는 Flask 기반 SaaS 웹 서비스의 고동시성 환경에서의 성능 최적화를 위해 Gunicorn Worker 모델을 Sync에서 Gevent로 변경하는 방안을 제시하고 그 효과를 실험적으로 검증하였다. Gevent의 Greenlet 기반 코루틴과 Monkey Patching을 활용하여 비동기 I/O를 구현한 결과, 동시 접속자 10명 이상의 환경에서 20% 이상의 응답 시간 개선을 달성하였다. 특히 50명 동시 접속 환경에서 78.1%의 성능 개선이 확인되었으며, 100명 환경에서도 안정적 처리가 가능함이 검증되어, 제한된 서버 자원으로 더 많은 사용자를 수용할 수 있는 비용 효율적인 최적화 방안임을 입증하였다.

본 연구의 핵심 기여는 워커 프로세스 수를 줄이면서도(10개 → 8개) worker_connections를 통해 동시 처리 능력을 대폭 향상시킨(10개 → 2,400개) 전략을 제시한 것이다. 이는 메모리 효율성과 고동시성을 동시에 달성할 수 있는 실용적인 접근 방법이다.

향후 연구 과제로는 다음을 제시한다. 첫째, Nginx 로드 밸런싱 설정 최적화를 통한 추가 성능 개선 방안을 탐색할 필요가 있다. 둘째, Redis 캐싱 전략 개선을 통해 데이터베이스 부하를 줄이고 응답 시간을 단축할 수 있다. 셋째, Kubernetes와 같은 컨테이너 오케스트레이션 플랫폼을 활용한 자동 스케일링 방안을 연구할 필요가 있다. 넷째,

CPU 집약적 작업과 I/O 집약적 작업을 분리하여 각각에 최적화된 워커 모델을 적용하는 하이브리드 아키텍처를 설계할 수 있다.

REFERENCES

[1] Benoît Chesneau, "Gunicorn Design Documentation", https://docs.gunicorn.org/en/stable/design.html, 2024.
[2] Denis Bilenko, "gevent Documentation", http://www.gevent.org/intro.html, 2024.
[3] PEP 3333, "Python Web Server Gateway Interface v1.0.1", Python Software Foundation, https://peps.python.org/pep-3333/, 2010.
[그림1] WSGI 아키텍처 및 Gunicorn 구조 (https://june-coder.tistory.com/53)
[그림2] Synchronous와 Asynchronous 처리 방식 비교 (https://www.koyeb.com/blog/introduction-to-synchronous-and-asynchronous-processing)
[그림3] event의 Greenlet 동작 메커니즘 (출처: https://www.tigera.io/blog/the-sharp-edges-of-gevent/)
[그림4] Docker 컨테이너 기반 웹 애플리케이션 배포 구조 (출처: https://akira3030.github.io/formacion/articulos/python-flask-gunicorn-docker.html)

저 자 소 개

이해인(정회원)
2006년 한밭대학교 산업경영학과 학사 졸업.
2013년 공주대학교 컴퓨터교육학과 석사 졸업.
2018년 공주대학교 컴퓨터교육학과 박사 수료.
<주관심분야 : 생상형 교육프로그램,인공지능, AI, ICT, 빅데이터 등>