백엔드 & 인프라 스터디 4 회차 문제

2025-10-25

작성자 : 김호중

상황

당신은 우리 회사의 핵심 데이터 처리 파이프라인을 담당하는 백엔드 개발자로 면접을 보고 있습니다. 이 시스템에는 log-processor 라는 커스텀 데몬(Daemon)이 있습니다. 이 데몬의 임무는 간단합니다.

실시간으로 /var/log/nginx/access.log 파일을 감시합니다.

로그 한 줄이 들어올 때마다, awk 를 사용해 특정 필드(예: 7 번째 필드인 URI)를 추출합니다.

이 URI 를 내부 API 로 전송하여 통계를 집계합니다.

이 서비스는 systemd 에 log-processor.service 라는 이름으로 등록되어 수개월간 잘 동작했습니다. 하지만 오늘 아침부터, API 서버의 통계 대시보드가 업데이트되지 않고 있습니다. 당신은 장애 대응을 위해 즉시 운영 서버에 SSH 로 접속했습니다.

문제

첫째, 당신이 가장 먼저 실행한 명령어는 systemctl status log-processor.service 입니다. 출력 결과, 놀랍게도 Active: active (running) 상태로 나옵니다. 분명히 서비스는 실행 중이지만 '동작하지(working)' 않는, 가장 까다로운 상황입니다. systemd 의 active (running) 상태가 실제로는 무엇을 의미하는지, 그리고 이것이 왜 서비스의 정상 동작을 보장하지 않는지 그 한계를 설명하시오. 이 모순적인 상황에서, 당신이 시스템의 실제 상태를 파악하기 위해 가장 먼저 확인할 다음 세 가지는 무엇이며, 그 이유는 무엇입 Gapsio?

둘째, 당신은 ps, top, htop 등을 통해 프로세스 목록을 확인했습니다. log-processor 부모 프로세스는 CPU 0%, Memory 0.1%로 거의 유휴(idle) 상태이지만, 그 자식 프로세스인 awk 가 'D' 상태(Uninterruptible Sleep)에 빠져있는 것을 발견했습니다. 'D' 상태가 무엇인지, 그리고 이 상태가 왜 일반적인 'S' (Interruptible Sleep) 상태보다 훨씬 심각한 문제인지 커널의 관점에서 설명해야 합니다. 이 awk 프로세스가 도대체 무엇을 기다리느라 'D' 상태에 빠졌을지, 현재 상황(로그 처리)과 연관 지어 가장 가능성 높은 가설을 제시하시오.

셋째, 당신은 log-processor 데몬 자체가 /var/log/ 디렉터리에 자신의 로그를 남기는 것을 확인했습니다. 이 로그 파일은 수 GB 에 달하며, "WARN"이나 "ERROR"가 아닌 "INFO" 레벨의 로그가 초당 수천 개씩 쌓여 원인 파악을 방해하고 있습니다. 당신은 이 거대한 로그 파일에서, 지난 1 시간 동안 발생한 로그 중, URI 경로(awk 로 추출한 7 번째 필드)에 /api/v1/health_check 가 아닌 모든 로그 라인에서, **IP 주소(1 번째 필드)만을 추출하여, 가장 많이 요청한 순서(Top 10)**로 집계해야 합니다. 이 작업을 수행하기 위한 grep, awk, sed 를 조합한 단 하나의 쉘 파이프라인 명령어를 어떻게 구성하시겠습니까? (정확한 구문보다, 각 도구를 어떤 역할로 조합했는지에 대한 논리적 흐름을 중점적으로 봅니다.)

넷째, 원인을 추적하던 중, 당신은 /usr/local/bin 에 있던 log-processor 바이너리가 누군가의 실수로 /etc/log-processor/ 디렉터리로 이동된 것을 발견했습니다. 리눅스 파일 시스템 표준(FHS)의 관점에서, /usr 디렉터리와 /etc 디렉터리의 근본적인 역할과 목적의 차이는 무엇입니까? 실행 가능한 바이너리를 /etc 에 두는 것이 왜 심각한 안티 패턴인지, 그리고 이 변경 사항이 어떻게 이번 장애와 간접적으로 연관될 수 있는지(예: SELinux/AppArmor 정책 문제) 당신의 통찰을 설명하시오.

다섯째, 모든 문제를 해결한 후, 당신은 이 log-processor.service 를 더 견고하게 만들기로 결정했습니다. 누군가 실수로 데몬의 실행 권한을 변경하여 서비스가 시작조차 못 하는 상황을 방지하고 싶습니다. chmod 755 /usr/local/bin/log-processor 와 chown root:root /usr/local/bin/log-processor 를 통해 소유권과 권한을 설정했습니다. 이 755 라는 숫자(Octal notation)가 'User', 'Group', 'Other' 각각에게 어떤 권한을 부여하며, '실행 파일'에 755 를, '설정 파일'에 644 를 부여하는 것이 왜 표준적인 보안 관행인지 그 이유를 논리적으로 설명해야 합니다.

상황 – 우리 프로젝트

당신이 제출한 경매 플랫폼의 설계 문서는 매우 인상적입니다. Redis Sorted Set 을 활용한 실시간 랭킹, Django Channels 와 Pub/Sub 을 통한 비동기 통신, Write-Behind 패턴을 통한 DB 부하 분산, 그리고 Saga 패턴을 통한 트랜잭션 관리까지, 현대적인 백엔드 아키텍처의 핵심 요소를 모두 정확히 짚어냈습니다.

하지만 설계 문서상의 아키텍처는 맑은 날씨에 그린 지도와 같습니다. 저는 당신이 폭풍우 속에서 이 배를 어떻게 운항할 것인지가 궁금합니다. 이제, 당신의 이 견고해 보이는 설계를 함께 스트레스 테스트해 보겠습니다.

문제

첫째: Write-Behind 패턴과 '진실의 순간'

당신의 설계는 응답 속도를 위해 Write-Behind 패턴을 채택했습니다. 즉, "입찰이 발생하면 Redis 에 먼저 기록하고, 백그라운드 태스크(Celery)로 MySQL 에 동기화한다"고 하셨습니다. 이는 매우 훌륭한 전략입니다.

시나리오: 경매 마감 1 분 전, 입찰이 폭주합니다. 사용자의 입찰 요청은 Redis 에 50ms 만에 기록되고, WebSocket 을 통해 "당신이 최고가 입찰자입니다!"라는 알림이 즉시 전송됩니다. 하지만 바로 그 순간, Celery 워커가 사용하는 메시지 브로커(예: RabbitMQ)가 3 분간 응답 불능 상태에 빠졌습니다.

경매는 예정대로 9 시에 마감되었습니다. Redis 에는 '사용자 A'가 100,000 원에 낙찰자로 기록되어 있습니다. 하지만 MySQL 의 마지막 기록은 3 분 전의 '사용자 B' (99,000 원)입니다.

질문:

경매가 종료된 9 시 00 분, 시스템의 'Source of Truth(진실의 원천)'는 무엇입니까? Redis 입니까, MySQL 입니까?

브로커가 복구되어 3 분간 밀렸던 500 개의 입찰 로그가 Celery 로 쏟아져 들어옵니다. 이미 '종료된' 경매에 대해 이 작업들을 어떻게 처리해야 합니까?

더 심각한 것은, '사용자 A'는 이미 WebSocket 을 통해 자신이 낙찰되었다는 알림을 받았습니다. 하지만 비즈니스 로직상 3 분 전의 '사용자 B'가 최종 낙찰자일 수도 있습니다. 이 데이터 불일치를 어떻게 감지하고, 어떻게 복구(reconcile)할 것이며, 이미 잘못된 기대를 하고 있는 '사용자 A'에게는 이 상황을 어떻게 설명하시겠습니까?

둘째: Redis Sorted Set 의 '공정성' 함정

당신의 핵심 입찰 로직은 ZADD auction:1 <price> <user_id>를 사용하여 Redis 의 Sorted Set 에 입찰가를 저장하는 것입니다. 이는 O(log N)의 환상적인 성능을 보장합니다.

시나리오: 9 시 00 분 00.100 초에 '사용자 A'가 100,000 원을 입찰했습니다. 정확히 0.2 초 뒤인 9 시 00 분 00.300 초에 '사용자 B'가 동일하게 100,000 원을 입찰했습니다.

질문:

두 입찰이 ZSET 에 ZADD auction:1 100000 "user_A"와 ZADD auction:1 100000 "user_B"로 저장되었을 때, ZSET 에서 최고 점수(Score)가 동점일 경우 멤버(Member)는 어떻게 정렬됩니까?

경매의 비즈니스 규칙은 "동일한 최고가일 경우, 먼저 입찰한 사람이 승리한다"입니다. 현재 당신의 ZSET 스키마가 이 선착순 공정성'을 보장합니까?

보장하지 못한다면, 이 공정성 규칙을 만족시키기 위해 ZSET 의 스코어(score) 또는 멤버 구성을 어떻게 변경하시겠습니까? 또한, 당신이 제안한 새로운 방식이 기존의 O(log N) 성능과 로직 복잡도에 어떤 영향을 미치는지 그 트레이드오프를 설명해 주십시오.

셋째: Saga 패턴의 '보상 트랜잭션'이라는 신기루

당신은 "입찰-결제-정산" 프로세스를 위해 Saga 패턴을 적용하겠다고 했습니다. 이는 긴 트랜잭션을 피하는 훌륭한 선택입니다.

시나리오: 경매가 종료되고 '사용자 A'가 낙찰되었습니다. Saga 가 시작됩니다.

(Orchestrator)가 AuctionService 에 "경매 종료" 이벤트를 보냅니다.

AuctionService 는 경매 상태를 CLOSED 로 변경하고, "결제 요청" 이벤트를 발행합니다.

PaymentService 가 이벤트를 받아, '사용자 A'의 등록된 카드로 결제를 시도합니다.

결제가 실패합니다.

질문:

Saga 의 핵심은 보상 트랜잭션(Compensating Transaction)입니다. AuctionService 가 이미 CLOSED 로 처리한 상태를 되돌리기 위한 보상 트랜잭션은 정확히 무엇입니까?

단순히 경매 상태를 다시 OPEN 으로 되돌리는 것이 올바른 비즈니스 로직일까요? 이미 수백 명의 다른 입찰자들은 떠났습니다.

만약 보상 트랜잭션이 '2 순위 입찰자에게 낙찰 시도'라면, 이는 또 다른 Saga 의 시작을 의미합니다. 만약 2 순위, 3 순위... 10 순위 입찰자까지 모두 결제에 실패하면 이 Saga 는 어떻게 종료됩니까?

Saga 패턴이 데이터베이스 레벨의 트랜잭션 롤백과 근본적으로 어떻게 다른지, 그리고 이 '결제 실패' 시나리오를 통해 당신이 깨달은 Saga 패턴의 진짜 비용(복잡성)**은 무엇인지 설명해 주십시오.

넷째: 시스템의 한계점과 '우아한 실패'

당신의 설계는 훌륭하지만, 모든 시스템에는 물리적인 한계가 있습니다.

시나리오: 경매 마감 1 분 전, 당신이 예상한 1 만 명이 아닌 100 만 명의 동시 사용자가 입찰을 시도합니다. 이는 정상적인 트래픽이 아닌 사실상의 DDoS 공격 수준입니다. 당신의 ALB, Nginx, Django, Redis, MySQL 로 구성된 시스템 전체가 처리 용량의 100 배에 달하는 요청을 받습니다.

질문:

이 상황에서 당신의 시스템은 어떻게 동작해야 합니까? "오토스케일링으로 모두 처리한다"는 비현실적인 답변 외에, 시스템이 '우아하게 실패(Fail Gracefully)'한다는 것은 구체적으로 무엇을 의미합니까?

시스템 전체가 다운되어 100 만 명 모두에게 504 Gateway Timeout 을 반환하는 최악의 상황을 막기 위해, 어느 계층(예: ALB, Nginx)에서 이 트래픽을 '차단(Shed Load)'하시겠습니까?

차단하기로 결정했다면, 어떤 기준으로 요청을 차단하시겠습니까? 무작위로 99%를 버리시겠습니까, 아니면 특정 기준(예: 악성 IP, 비로그인 사용자)으로 요청을 선별하시겠습니까?

살아남은 1%의 사용자와, 차단된 99%의 사용자에게는 각각 어떤 HTTP 상태 코드와 응답을 보내야, 사용자의 재시도 폭풍(Retry Storm)을 유발하지 않고 상황을 가장 잘 제어할 수 있을까요?

상황

당신은 사용자가 대용량 보고서(수백 MB)를 생성하고 다운로드할 수 있는 B2B SaaS 플랫폼의 백엔드 개발자로 면접을 보고 있습니다. 시스템은 Nginx, Gunicorn, 그리고 Django 로 구성된 표준적인 아키텍처를 따르고 있습니다.

서비스 런칭 후, 대부분의 기능은 정상적으로 동작하지만, 특정 고객사로부터 "보고서 다운로드가 1 분을 넘어가면 무조건 실패한다"는 치명적인 불만이 접수되었습니다. 당신은 이 문제를 재현하는 데 성공했습니다.

사후 분석 결과, 상황은 다음과 같습니다.

증상: 정확히 60 초가 경과하는 시점에, 사용자는 HTTP 504 Gateway Timeout 오류를 받습니다.

애플리케이션 로그: Django 로그에는 어떠한 오류도 기록되지 않았습니다. Django 는 20 초 만에 보고서 생성을 완료하고 HttpResponse 를 반환한 것으로 기록되어 있습니다.

Nginx 로그: Nginx 의 error.log 에는 upstream request timeout 이라는 기록이, access.log 에는 504 상태 코드가 기록되어 있습니다.

서버 리소스: top 으로 확인한 결과, 장애 발생 순간에도 서버의 CPU 와 메모리는 매우 여유로운 상태였습니다.

Gunicorn 설정: Gunicorn 은 기본 'sync' 워커를 사용하고 있으며, 워커의 타임아웃은 300 초(--timeout 300)로 넉넉하게 설정되어 있습니다.

당신에게 주어진 임무는, 이 흩어진 단서들을 바탕으로 미스터리한 504 에러의 근본 원인을 진단하고, 재발을 방지할 수 있는 견고한 해결책을 제시하는 것입니다.

문제

첫째, 모든 단서가 가리키는 진범은 단 하나입니다. Django 는 20 초 만에 응답을 완료했는데도 불구하고, 클라이언트는 60 초 후에 타임아웃을 경험했습니다. 이 40 초의 차이는 어디에서 발생한 것이며, Nginx 가 upstream request timeout 을 기록한 진짜 이유는 무엇입니까? 이 문제의 근본 원인을 Nginx 의 리버스 프록시 동작 방식과 Gunicorn 'sync' 워커의 본질적인 한계, 그리고 느린 클라이언트(Slow Client)'라는 개념을 엮어서 심층적으로 설명하시오.

둘째, 한 주니어 개발자가 "Gunicorn 의 --timeout 을 60 초로 낮추면 Nginx 와 설정이 맞으니 해결되지 않을까요?"라고 제안했습니다. 이 제안이 왜 문제의 본질을 완전히 잘못 파악한 것이며, 오히려 상황을 악화시킬 수 있는지 설명하시오. 또한, 또 다른 개발자가 "Nginx 를 아예 없애고 Gunicorn 을 80 번 포트에서 직접 서비스합시다. 그럼 이 복잡한 프록시 문제가 사라질 겁니다."라고 제안했습니다. 이 제안이 왜 더 심각한 재앙을 초래할 수 있는지, Gunicorn 의 'pre-fork' 동기 워커 모델과 Nginx 의 '이벤트 기반 비동기' 아키텍처의 근본적인 차이점을 들어 반박하시오.

셋째, 이 문제를 해결하기 위한 두 가지 서로 다른 아키텍처적 접근법을 제시하고, 각각의 명확한 트레이드오프를 비교하시오.

접근법 A (Nginx 튜닝): Nginx 의 proxy_buffering 관련 지시어(directives)를 조정하여 문제를 해결하는 방안. 이 방식이 어떻게 Gunicorn 워커를 즉시 해제시키고, 대신 어떤 자원(메모리/디스크)을 희생시키는지 그 원리를 설명하시오

접근법 B (Gunicorn 튜닝): Gunicorn 의 워커 타입을 'sync'가 아닌 'gevent'나 'asyncio'와 같은 비동기(async) 워커로 변경하는 방안. 이 방식이 어떻게 Nginx 의 버퍼링 없이도 문제를 해결할 수 있는지, 그리고 이로 인해 애플리케이션 코드베이스에 어떤 새로운 복잡성(예: 'monkey-patching', 비동기 DB 드라이버)이 요구되는지 설명하시오.

넷째, 당신은 Gunicorn 의 마스터 프로세스에 SIGHUP 시그널을 보내면, 서비스 중단 없이(Zero-Downtime) 애플리케이션 코드를 리로드할 수 있다는 것을 알고 있습니다. 이 '우아한 재시작(Graceful Restart)'이 정확히 어떤 내부 메커니즘으로 동작하는지, 마스터 프로세스와 기존 워커 프로세스, 그리고 새로 생성된 워커 프로세스의 생명주기(Lifecycle) 관점에서 단계별로 설명하시오. 만약 한 워커가 5 분짜리 긴 작업을 처리하는 도중에 SIGHUP 시그널을 받는다면, 이 워커는 어떻게 됩니까?

다섯째, Nginx 와 Gunicorn 의 통신 방식에는 TCP 소켓(127.0.0.1:8000)을 사용하는 것과 유닉스 도메인 소켓(/run/gunicorn.sock)을 사용하는 것이 있습니다. 대부분의 가이드는 유닉스 소켓을 권장합니다. 단순히 '더 빠르다'는 표면적인 이유를 넘어, 이 두 방식의 근본적인 차이점이 무엇인지 OSI 7 계층의 관점에서 설명하시오. 유닉스 소켓이 어떻게 네트워크 스택의 오버헤드를 우회하는지, 그리고 파일 시스템 권한을 통해 어떻게 더 강력한 보안을 제공할 수 있는지 당신의 통찰을 제시해야 합니다.