Django 백엔드 스터디 5 회차 문제

2025-08-27

작성자 : 김호중

백엔드 - 논리적사고문제: 컨테이너 오케스트레이션 도입

한 소셜 커머스 플랫폼은 초기에 docker-compose 를 사용하여 단일 서버에서 웹 애플리케이션, 백그라운드 워커, 데이터베이스 등 여러 컨테이너를 운영했습니다. 서비스가 빠르게 성장하면서 일일 할인 이벤트 시간에는 트래픽이 평소의 10 배 이상으로 급증했고, 단일 서버의 한계에 부딪혔습니다.

엔지니어링팀은 급한 대로 서버 사양을 높이고(Scale-up), 이벤트 시간에 맞춰 수동으로 웹 애플리케이션 컨테이너 수를 늘리는(docker-compose up --scale web=10) 방식으로 대응했습니다. 하지만 이 과정에서 다음과 같은 문제들이 발생했습니다.

반응성 저하: 트래픽이 급증한 것을 인지하고 수동으로 스케일링하는 데 시간이 걸려, 피크 타임 초반에 서비스 장애가 발생하는 경우가 잦았습니다.

취약한 가용성: 운영 중인 서버 한 대에 하드웨어 장애가 발생하자, 해당 서버에서 실행되던 모든 컨테이너가 멈춰버려 서비스 전체가 중단되는 사고를 겪었습니다.

잦은 배포 오류: 새로운 기능을 배포할 때마다 기존 컨테이너를 내리고 새 컨테이너를 올리는 과정에서 수 초간의 서비스 다운타임이 발생했으며, 가끔 설정 실수로 배포가 실패하여 이전 버전으로 되돌리는 데 큰 어려움을 겪었습니다.
결국, 팀은 이런 문제를 근본적으로 해결하기 위해 Kubernetes 도입을 결정했습니다.

문제

기존 docker-compose 환경의 문제점들을 Kubernetes 의 핵심 기능과 연결하여, Kubernetes 도입이 기술적으로 왜 필연적인 선택이었는지 3 가지 관점(탄력성, 가용성, 배포 안정성)에서 설명하시오.

수동 스케일링의 한계를 극복하기 위해 Kubernetes 의 **HPA(Horizontal Pod Autoscaler)**를 어떻게 구성할 수 있을까요? CPU 사용률을 기준으로 하는 HPA 의 동작 원리와, 설정 시 고려해야 할 핵심 파라미터(minReplicas, maxReplicas, targetCPUUtilizationPercentage)의 역할을 설명하시오.

서버 장애 시 서비스 전체가 중단되었던 문제를 Kubernetes 는 어떻게 해결하나요? Pod, Node, ReplicaSet 의 관계를 통해 Kubernetes 의 '자동 복구(Self-healing)' 메커니즘을 설명하시오.

배포 시 다운타임이 발생했던 문제를 해결하기 위한 무중단 배포(Zero-downtime Deployment) 전략을 설명하시오. Kubernetes 의 Deployment 오브젝트가 사용하는 '롤링 업데이트(Rolling Update)' 방식이 어떻게 사용자의 서비스 중단 없이 안전하게 신규 버전을 배포하는지 그 과정을 단계별로 서술하시오.

백엔드 - 논리적사고문제: Infrastructure as Code (IaC) 도입

한 핀테크 스타트업의 인프라팀은 AWS 클라우드 위에 서비스를 구축했습니다. 초기에는 소수의 엔지니어가 AWS 웹 콘솔에 직접 로그인하여 필요한 리소스(VPC, EC2, RDS, S3 등)를 수동으로 생성하고 설정했습니다.
회사가 성장하고 서비스가 복잡해지면서 이 방식은 심각한 문제들을 낳기 시작했습니다.

환경 불일치: 개발 환경과 운영 환경의 네트워크 설정(Security Group)이 미세하게 달라, 개발 환경에서는 잘 동작하던 기능이 운영 환경에 배포하면 실패하는 문제가 반복되었습니다. 원인을 찾는 데에만 수일이 소요되기도 했습니다.

변경 이력 부재: 누군가 급하게 운영 DB 의 설정을 변경했지만, 누가, 언제, 왜 변경했는지 아무런 기록이 남아있지 않아 보안 감사를 통과하지 못했습니다.

느린 확장 속도: 새로운 파트너사를 위한 독립적인 테스트 환경을 구축해달라는 요청에, 20 개가 넘는 리소스를 일일이 수동으로 생성하고 연결하느라 꼬박 일주일이 걸렸습니다.

재해 복구의 어려움: 한 리전(Region)에 장애가 발생하여 다른 리전에 인프라를 긴급 복구해야 하는 상황을 가정하고 모의 훈련을 진행했지만, 수백 개의 설정을 정확히 재현하는 것은 사실상 불가능에 가깝다는 결론을 내렸습니다.

이러한 문제들을 해결하기 위해, 인프라팀은 Terraform 을 도입하여 모든 인프라를 코드로 관리하기로 결정했습니다.

문제

AWS 웹 콘솔을 이용한 수동 관리 방식과 비교했을 때, Terraform 과 같은 IaC(Infrastructure as Code)를 도입해야 하는 결정적인 이유를 위 시나리오의 문제점(일관성, 추적성, 생산성, 재현성)과 연결하여 4 가지 측면에서 설명하시오.

Terraform 의 핵심 개념인 **'상태 파일(State File)'**은 어떤 역할을 하며, 왜 중요한가요? 만약 두 명의 엔지니어가 동시에 각자의 컴퓨터에서 인프라 변경 작업을 진행할 때 발생할 수 있는 치명적인 문제를 제시하고, 이를 방지하기 위한 '원격 상태 저장소(Remote Backend)와 상태 잠금(State Locking)'의 필요성을 설명하시오.

모든 환경(개발, 스테이징, 운영)의 인프라 코드를 하나의 거대한 .tf 파일에 작성했더니 관리가 어려워졌습니다. 코드의 재사용성을 높이고 환경별 차이를 효율적으로 관리하기 위해 Terraform 의 **모듈(Module)과 워크스페이스(Workspace)**를 어떻게 활용할 수 있을지 구체적인 리팩토링 전략을 제시하시오.

한 주니어 엔지니어가 실수로 운영 DB 를 삭제하는 Terraform 코드를 작성했습니다. 이러한 재앙을 방지하기 위해 Terraform 이 제공하는 안전장치는 무엇인가요? terraform plan 과 terraform apply 명령어의 역할을 설명하고, 코드 리뷰와 plan 결과물 검토가 왜 팀의 필수적인 워크플로우가 되어야 하는지 논하시오.

TCP/UDP 헤더 구조 기반 논리적 사고 문제

한 게임 개발사에서 실시간 1:1 대전 격투 게임을 개발하고 있습니다. 이 게임에서는 플레이어의 캐릭터 위치, 동작, 타격 판정 등 매초 수십 개의 작은 데이터 패킷을 상대방에게 빠르고 지속적으로 전송해야 합니다.

팀의 주니어 개발자가 "모든 패킷이 정확하게 전달되어야 게임이 성립하니, 신뢰성을 보장하는 TCP 를 사용해야 한다"고 강력하게 주장합니다. 하지만 시니어 개발자는 "그 주장은 TCP 헤더의 구조와 실시간 게임의 특성을 제대로 이해하지 못한 것"이라며, 오히려 UDP 가 더 적합한 선택이라고 반박합니다.

문제

헤더 오버헤드 관점: TCP 와 UDP 의 최소 헤더 크기를 비교하고, TCP 헤더에만 존재하는 필드 중 (Sequence Number, Acknowledgement Number, Window Size)가 실시간 격투 게임에서 불필요한 오버헤드가 되는 이유를 '데이터-헤더 비율'의 비효율성과 연관 지어 설명하시오.

신뢰성의 역설: 주니어 개발자가 장점이라고 생각한 '신뢰성 보장'이 실시간 게임에서는 오히려 치명적인 단점이 될 수 있습니다. TCP 헤더의 Sequence Number 와 Acknowledgement Number 를 이용한 재전송 및 순서 보장 메커니즘이 어떻게 지연 시간(Latency)을 유발하여, 이미 지나간 0.1 초 전의 캐릭터 위치 정보를 뒤늦게 수신하는 '끔찍한 경험'을 만들어내는지 설명하시오.

흐름 제어의 부적절성: TCP 헤더의 Window Size 필드는 송신자와 수신자 간의 데이터 흐름을 조절하는 역할을 합니다. 이 메커니즘이 왜 게임 서버처럼 다수의 클라이언트와 동시에, 각기 다른 네트워크 상태로 통신해야 하는 환경에 부적합한지 설명하고, 차라리 UDP 를 기반으로 **애플리케이션 레벨에서 자체적인 전송률 제어(Rate Control)**를 구현하는 것이 더 유연한 이유를 논하시오.

유일하게 유용한 필드: 시니어 개발자는 "TCP 헤더의 기능 대부분이 불필요하지만, Checksum 필드만큼은 UDP 에서도 반드시 사용해야 한다"고 덧붙였습니다. UDP 헤더에도 동일하게 존재하는 이 필드가, 신뢰성을 포기하는 UDP 통신에서 왜 여전히 중요한 역할을 하는지 데이터 무결성 관점에서 설명하시오.

네트워크 심화 통찰 기반 논리적 사고 문제

글로벌 OTT 서비스를 운영하는 한 회사가 동남아시아 시장에 새롭게 진출했습니다. 싱가포르에 대규모 리전(Region) 데이터센터를 구축했지만, 현지 사용자들로부터 "영상 로딩이 너무 느리고, 재생 중 끊김이 잦다"는 불만이 폭주했습니다.

기술팀이 분석한 결과, 싱가포르 데이터센터 자체의 서버 성능이나 회선 대역폭에는 문제가 없었습니다. 진짜 원인은 데이터센터에서 사용자들의 스마트폰까지 이어지는 '라스트 마일(Last Mile)' 구간이었습니다. 이 구간은 유선망에 비해 지연 시간이 길고(High Latency), 패킷 손실률이 높은(Lossy) 불안정한 무선 이동통신망이 대부분이었습니다. 특히, TCP 의 동작 방식이 이 환경에서 성능을 크게 저하시키고 있었습니다.

문제

병목 현상의 재정의: 기술팀은 사용자의 ping 시간(ICMP)이 높은 것을 확인했습니다. 하지만 이것만으로는 영상 로딩 시간(L7)이 왜 몇 초씩 걸리는지 완전히 설명할 수 없습니다. TCP 3-way handshake 와 TLS handshake 과정을 포함하여, 사용자가 '재생' 버튼을 눌렀을 때 실제 영상 데이터(HTTP Payload)의 첫 바이트가 도착하기까지 네트워크상에서 어떤 **추가적인 왕복(Round-Trip)**들이 발생하는지 설명하시오.

TCP 의 오해와 함정: 전통적인 TCP 혼잡 제어 알고리즘(예: CUBIC)은 패킷 손실을 '네트워크 혼잡'의 신호로 간주합니다. 이 가정이, 실제로는 망이 혼잡하지 않아도 무선 신호 품질 때문에 패킷 손실이 잦은 모바일 환경에서 왜 치명적인 성능 저하를 일으키는지 설명하시오. (Hint: 혼잡 윈도우(Congestion Window) 크기 변화와 연관 지어 서술)

아키텍처적 해결책 (CDN/Edge): 회사는 이 문제를 해결하기 위해 각 국가별 통신사 망 내부에 소규모 캐시 서버, 즉 CDN 엣지(Edge) PoP 을 구축하기로 결정했습니다. 이 아키텍처가 어떻게 '라스트 마일' 문제를 완화하는지 다음 세 가지 측면에서 설명하시오.

TCP 연결의 분리: 사용자와 엣지, 그리고 엣지와 원본 데이터센터 간의 TCP 연결이 어떻게 분리되고, 이것이 사용자 경험을 어떻게 개선하는가?
콘텐츠 캐싱: 자주 보는 영상 콘텐츠를 엣지에 캐싱하는 것의 효과.
프로토콜 최적화: 엣지와 원본 서버 사이의 안정적인 '미들 마일' 구간에서만 TCP 대신 더 공격적인 커스텀 전송 프로토콜을 사용할 경우의 이점.

프로토콜의 진화 (HTTP/3): 만약 아키텍처 변경 없이 프로토콜만으로 이 문제를 개선해야 한다면, TCP 기반의 HTTP/2 대신 **UDP 기반의 QUIC(HTTP/3)**를 도입하는 것이 왜 근본적인 해결책이 될 수 있을까요? QUIC 이 가진 **'Head-of-Line Blocking 제거'**와 '0-RTT 연결 재개' 기능이 불안정한 모바일 네트워크 환경에서 어떤 혁신적인 이점을 제공하는지 심층적으로 설명하시오.