배포 중 롤백 안전 보장

Sandeep Pokkunuri

Amazon에서 저희가 솔루션의 구축하는 방법에 대한 안내 원칙 중 하나는 출구가 1개인 방법은 피하자입니다. 다시 말해, 확장하거나 되돌리기 어려운 선택은 피하자는 뜻입니다. 저희는 제품, 기능, API 및 백엔드 시스템 설계에서 배포에 이르기까지 소프트웨어 개발의 모든 단계에서 이 원칙을 고수합니다. 이 글에서는 소프트웨어 배포에 이 원칙을 적용하는 방법에 대해 설명하고자 합니다.

배포는 소프트웨어 환경을 한 상태(버전)에서 다른 상태로 이전합니다. 이러한 단계에서는 소프트웨어가 완벽하게 작동할 수 있습니다. 하지만 정방향(업그레이드나 롤포워드) 또는 역방향(다운그레이드 또는 롤백)으로 전이하는 도중이나 이후에 소프트웨어가 올바르게 작동하지 않을 수 있습니다. 소프트웨어가 잘 작동하지 않으면 서비스 중단으로 이어지고, 이로 인해 고객 측에서 서비스가 불안정해질 수 있습니다. 이 글에서는 소프트웨어의 두 버전 모두가 예상대로 잘 작동한다고 가정합니다. 집중적으로 짚어볼 내용은, 배포 중 롤포워드나 롤백으로 인해 오류가 발생하지 않도록 보장하는 방식에 관한 부분입니다.

소프트웨어의 새 버전을 출시하기 전에 저희는 기능, 동시성, 성능, 확장성 및 다운스트림 장애 처리와 같은 여러 측면을 고려하여 베타 또는 감마 테스트 환경에서 소프트웨어를 테스트합니다. 이러한 테스트는 새 버전에서 문제를 파악하고 수정하는 데 도움이 됩니다. 그러나 항상 이것만으로 성공적인 배포를 보장할 수는 없습니다. 프로덕션 환경에서는 예상치 못한 상황이나 최적이 아닌 소프트웨어 동작이 나타날 수 있습니다. Amazon에서는 배포 롤백이 고객에게 오류를 발생시킬 수 있는 상황에 처하는 상황을 피하려고 합니다. 이 상황을 피하려면 모든 배포 전에 롤백에 완벽하게 준비해야 합니다. 이전 버전에서 사용한 기능 중단이나 오류 없이 롤백할 수 있는 소프트웨어 버전을 역호환 가능이라고 합니다. 저희는 저희 소프트웨어가 모든 버전에서 역호환 가능하도록 계획하고 이를 검증합니다.

Amazon의 소프트웨어 업데이트 방식에 대해 자세히 설명하기 전에 먼저 독립형 및 분산 소프트웨어 배포 사이의 몇 가지 차이점에 대해 이야기하겠습니다.

독립형 및 분산 소프트웨어 배포 비교

하나의 디바이스에서 하나의 프로세스로 실행되는 독립형 소프트웨어에 대한 배포는 원자성 특징을 띕니다. 소프트웨어의 두 가지 버전은 절대로 동시에 실행되지 않습니다. 독립형 소프트웨어가 상태를 유지하면 새 버전은 이전 버전이 쓴(직렬화) 데이터를 읽어야(직렬화 해제)하고, 아니면 그 반대로 수행해야 합니다. 이 조건을 만족하면 배포는 롤백 및 롤포워드 작업에 대해 안전합니다.

분산 시스템에서 배포는 더욱 복잡해집니다. 배포는 가용성이 영향을 받지 않도록 업데이트를 통해 수행됩니다. 새 버전은 다른 호스트가 계속 요청을 처리할 수 있도록 즉시 호스트의 한 하위 집합을 롤아웃합니다. 일반적으로 이러한 호스트는 RFC(원격 프로시저 호출) 또는 공유 영구 상태(예: 메타데이터 또는 체크포인트)를 통해 서로 통신합니다. 이러한 통신 또는 공유 상태로 추가 과제가 생길 수 있습니다. 쓰는 쪽과 읽는 쪽은 소프트웨어의 서로 다른 버전을 실행할 수 있습니다. 그 결과 데이터를 다르게 해석할 수 있습니다. 읽는 쪽은 데이터를 함께 읽는 데 실패할 수 있고, 이로 인해 가동 중단이 발생할 수 있습니다.

프로토콜 변경 관련 문제

롤백할 수 없는 가장 일반적인 이유는 프로토콜 변경이라는 사실을 깨달았습니다. 예를 들어, 데이터를 디스크에 지속시키는 동안 데이터 압축을 시작하는 코드 변경을 가정합니다. 새 버전이 압축된 데이터를 쓴 후에는 롤백은 가능한 옵션이 아닙니다. 이전 버전은 디스크에서 읽은 후 데이터를 압축 해제해야 함을 알지 못합니다. 데이터가 BLOB 또는 문서 저장소에 저장된 경우 다른 서버는 배포가 진행 중이어도 데이터를 읽지 못합니다. 이 데이터가 두 프로세스나 서버 사이에서 전달되면 받는 쪽은 데이터를 읽지 못합니다.

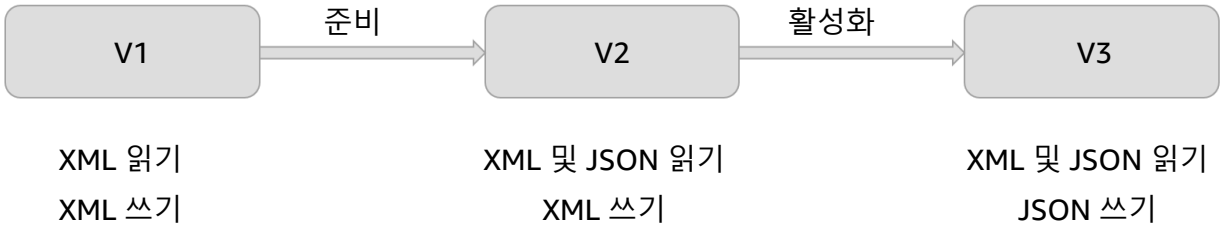
때로는 프로토콜 변경은 매우 사소한 것일 수 있습니다. 예를 들어, 연결에서 비동기식으로 통신하는 두 서버를 고려합니다. 두 서버가 모두 활성화되었음을 서로 인지하기 위해 5초마다 서로 하트비트를 전송하기로 합의했습니다. 서버가 약정한 시간 내 하트비트를 확인하지 못하면 다른 서버가 중단되었다고 가정하고 연결을 닫습니다.

이제 하트비트 기간을 10초로 늘린 배포를 고려합니다. 코드 커미트는 사소해보입니다. 숫자만 바뀐 것뿐이니까요. 하지만 이제는 롤포워드 및 롤백 모두에 대해 안전하지 않습니다. 배포 중 새 버전을 실행하는 서버가 10초마다 하트비트를 전송합니다. 결과적으로 이전 버전을 실행하는 서버는 5초 넘게 하트비트를 확인하지 못하며 새 버전을 실행하는 서버와의 연결을 종료합니다. 대형 플릿에서는 여러 연결에서 이러한 상황이 발생할 수 있으며 이로 인해 가용성이 떨어집니다.

이러한 사소한 변화로 인해 코드를 읽거나 문서를 설계하여 분석할 때 어려움이 따릅니다. 따라서 저희는 롤포워드와 롤백에 대해 각 배포가 안전하다는 점을 명시적으로 검증합니다.

2단계 배포 기술

안전하게 롤백할 수 있도록 보장하기 위해 사용하는 한 가지 방법은 보통 2단계 배포라는 기술을 사용하는 것입니다. Amazon Simple Storage Service(Amazon S3)에서 데이터를 관리하는 서비스(쓰기, 읽기)에 대해 다음과 같은 가정을 적용한 시나리오를 고려합니다. 서비스는 가용성 및 확장성을 위해 여러 가용 영역에 걸쳐 서버 플릿에서 실행됩니다.
현재 서비스는 데이터를 지속시키기 위해 XML을 사용합니다. 버전 V1에서 다음 다이어그램과 같이 모든 서버는 XML을 쓰고 읽습니다. 비즈니스상의 이유로 저희는 JSON 형식으로 데이터를 지속시킵니다. 한 배포에서 이와 같이 변경하면 변경 사항을 선택하는 서버는 JSON으로 씁니다. 하지만 다른 서버는 아직 JSON을 읽는 방법을 모릅니다. 이 상황 때문에 오류가 발생합니다. 따라서 2개 부분으로 이러한 변경을 구분하고 2단계 배포를 수행합니다.



이전 다이어그램과 같이, 첫 번째 단계를 준비 단계라고 합니다. 이 단계에서는 XML 외에도 JSON을 읽도록 모든 서버를 준비하지만, 서버는 버전 V2를 배포하여 계속 XML을 씁니다. 이 변경 사항은 운영 관점에서 어떠한 것도 변경하지 않습니다. 모든 서버는 계속 XML을 읽고 모든 데이터는 여전히 XML로 씁니다. 이 변경 사항을 롤백하기로 결정한 경우 서버는 JSON을 읽을 수 없는 상태로 돌아갑니다. 어떠한 데이터도 아직 JSON으로 쓰여지지 않았기 때문에 전혀 문제가 되지 않습니다.

이전 다이어그램과 같이, 두 번째 단계를 활성화 단계라고 합니다. 이 단계에서는 버전 V3를 배포하여 쓰기에 대해 JSON 형식을 사용하도록 서버를 활성화합니다. 각 서버가 이 변경 사항을 선택하면 JSON으로 쓰기 시작합니다. 아직 이 변경 사항을 선택하지 않은 서버도 첫 번째 단계에서 준비되었기 때문에 JSON을 읽을 수 있습니다. 이 변경 사항을 롤백하기로 결정한 경우 일시적으로 활성화 단계였던 서버가 쓴 모든 데이터는 JSON입니다. 활성화 단계가 아니었던 서버가 쓴 데이터는 XML입니다. 이러한 상황도 V2에서 확인할 수 있듯이 서버는 롤백 후에도 XML 및 JSON을 모두 읽을 수 있으므로 아무런 문제가 없습니다.
이전 다이어그램에서는 XML에서 JSON으로 직렬화 형식의 변경을 보여주지만, 일반 기술은 앞서 프로토콜 변경 섹션에서 설명한 모든 상황에 적용 가능합니다. 예를 들어, 서버 사이의 하트비트 기간이 5초에서 10초로 늘려야 하는 이전 시나리오를 다시 살펴보겠습니다. 준비 단계에서 모든 서버가 5초에 한 번 하트비트를 계속 전송하지만, 모든 서버가 예상 하트비트 기간을 10초로 완화하도록 설정할 수 있습니다. 활성화 단계에서는 10초마다 한 번으로 빈도를 변경합니다.

2단계 배포 시 주의 사항

이제 2단계 배포 기술을 따를 때 주의할 사항에 대해 설명하고자 합니다. 이전 섹션에서 설명한 시나리오 예를 언급할 때도 이미 이러한 주의 사항이 대부분의 2단계 배포에 적용되어 있습니다.

많은 배포 도구에서는 사용자가 최소의 호스트가 변경 사항을 선택하고 정상 상태로 보고하는 경우 배포에 성공했다고 생각합니다. 예를 들어, AWS CodeDeploy에는 minimumHealthyHosts라고 하는 배포 구성이 있습니다.

2단계 배포 예에서 중요한 한 가지 가정은, 첫 번째 단계가 끝나면 모든 서버가 XML 및 JSON을 읽도록 업그레이드되었다는 점입니다. 둘 이상의 서버가 첫 번째 단계 중에 업그레이드되지 못하면 두 번째 단계 도중과 이후에 데이터를 읽지 못합니다. 따라서 모든 서버가 준비 단계에서 변경 사항을 선택했는지 명시적으로 검증합니다.

Amazon DynamoDB에서 작업할 때 저희는 여러 마이크로서비스에 걸쳐 있는 방대한 양의 서버 사이에서 통신 프로토콜을 변경하기로 한 적이 있습니다. 저는 모든 서버가 먼저 준비 단계에 도달한 후 활성화 단계를 진행하도록 모든 마이크로서비스 사이에서 배포를 조정했습니다. 이때 주의 사항으로 각 단계 끝에서 모든 단일 서버에서 배포가 성공했음을 명시적으로 검증했습니다.

2단계 각각이 롤백에 안전한 동안에는 두 변경 사항을 롤백할 수 없습니다. 이전 예에서는 활성화 단계가 끝날 때 서버는 JSON으로 데이터를 씁니다. 준비 및 활성화 변경 이전에 사용할 소프트웨어 버전은 JSON을 읽는 방법을 모릅니다. 따라서 주의 사항으로 준비와 활성화 단계 사이에서 경과하는 시간을 두기로 했습니다. 이 시간을 굽는 시간이라고 불렀으며, 이 기간은 보통 며칠 정도가 걸립니다. 그리고 다른 버전으로 롤백하지 않아도 되는 상황이 되기까지 기다립니다.

활성화 단계 이후에도 소프트웨어의 XML 읽기 기능을 안전하게 제거할 수 없습니다. 준비 단계 이전에 쓴 모든 데이터는 XML이기 때문에 제거하기에 안전하지 않습니다. 모든 단일 객체가 JSON으로 다시 쓸 수 있도록 보장한 후에만 XML 읽기 기능을 제거할 수 있습니다. 이 프로세스를 다시 채우기라고 합니다. 이 경우 서비스가 데이터를 읽고 쓰는 동안 동시에 실행할 수 있는 추가 도구가 필요할 수도 있습니다.

직렬화 모범 사례

대부분의 소프트웨어는 데이터 직렬화(네트워크에서 전송하거나 지속하는지 여부)를 포함합니다. 소프트웨어가 발전하면서 직렬화 논리도 변경되었습니다. 새 필드 추가에서 완전한 형식 변화에 이르기까지 변경 사항은 다양합니다. 수년 동안 저희는 직렬화에 대해 따라야 하는 몇 가지 모범 사례를 구축했습니다.

- 일반적으로 사용자 지정 직렬화 형식 개발을 피합니다.

  사용자 지정 직렬화의 초기 논리는 사소해보이고, 심지어 성능 개선 측면도 있어 보입니다. 하지만 이후 형식이 반복되면 JSON, Protocol Buffers, Cap'n Proto 및 FlatBuffers와 같은 잘 구축된 프레임워크가 이미 해결한 과제들이 생겨날 수 있습니다. 이러한 프레임워크는 적절히 사용하면 이스케이프, 역호환성 및 속성 존재 추적(즉, 필드가 명시적으로 설정되었는지, 아니면 내재적으로 기본값을 지정하는지 여부)과 같은 안전 기능을 제공합니다.

- 변경할 때마다 명시적으로 직렬화 항목에 개별 버전을 지정합니다.

  소스 코드나 버전 구축과는 별개로 진행합니다. 또한 직렬화된 데이터나 메타데이터에서 직렬화 항목 버전을 저장합니다. 오래된 직렬화 항목 버전도 새 소프트웨어 계속 작동합니다. 일반적으로 읽거나 쓴 데이터 버전에 대한 지표를 생성하는 데 유용합니다. 그리고 오류가 있으면 운영자에게 가시성과 문제 해결 정보도 제공합니다. 이 모두가 RPC 및 AP 버전에도 적용됩니다.

- 제어할 수 없는 데이터 구조의 직렬화를 피합니다.

  예를 들어, 리플렉션을 사용하여 Java 콜렉션 객체를 직렬화할 수 있습니다. 하지만 JDK를 업그레이드하려고 하면 이러한 클래스의 기본 구현이 변경되어 직렬화 해제에 실패할 수 있습니다. 이 위험은 팀에서 공유하는 라이브러리의 클래스에도 해당됩니다.
- 일반적으로 알 수 없는 속성의 존재를 허용하도록 직렬화 항목을 설계합니다.

  가능한 경우 직렬화 항목은 알 수 없는 속성을 유지하면서 데이터를 다시 씁니다. 이를 통해 소프트웨어의 새 버전을 실행하는 서버가 데이터에 새 속성을 포함하는 동시에 직렬화를 수행해도 이전 버전을 실행하는 서버는 속성을 지우지 않고 동일한 데이터를 업데이트합니다. 따라서 2단계 배포는 필요하지 않습니다.

저희의 많은 모범 사례와 같이, 저희의 지침이 모든 애플리케이션과 시나리오에 다 해당되는 것은 아니라는 주의 사항과 함께 이를 공유합니다.

변경이 롤백에 대해 안전한지 확인

일반적으로 저희는 소프트웨어 변경이 업그레이드-다운그레이드 테스트라고 하는 기능을 통해 롤포워드와 롤백에 안전한지 명시적으로 검증합니다. 이 프로세스를 위해 프로덕션 환경을 대변하는 테스트 환경을 설정합니다. 수년 간 저희는 테스트 환경을 설정할 때 몇 가지 피해야 하는 패턴을 식별해왔습니다.

프로덕션에서 변경을 배포하면 변경이 테스트 환경에서 모든 테스트를 통과했어도 오류를 발생시키는 상황을 봐오곤 했습니다. 한 번은 테스트 환경의 서비스가 각각 하나의 서버를 보유했었습니다. 따라서 모든 배포는 원자 배포였기 때문에, 소프트웨어의 다른 버전을 동시에 실행할 가능성이 배제되었습니다. 지금은 테스트 환경에 프로덕션 환경만큼 많은 트래픽이 없어도 각 서비스 뒤의 여러 가용 영역에 있는 여러 서버를 사용합니다. 프로덕션에서만큼의 수준이지요. Amazon에서는 절약을 좋아하지만, 품질과 관련해서는 아닙니다.

또 다른 경우에, 테스트 환경에 여러 서버가 있었습니다. 하지만 배포는 테스트를 가속화하도록 한 번에 모든 서버에서 이루어졌습니다. 이 접근 방식은 소프트웨어의 이전 버전과 새 버전을 동시에 실행하지 않도록 합니다. 롤포워드 문제는 감지되지 않았습니다. 이제 저희는 모든 테스트 및 프로덕션 환경에서 동일한 배포 구성을 사용합니다.

마이크로서비스 사이의 조정을 포함하는 변경 사항의 경우 테스트 및 프로덕션 환경의 마이크로서비스에서 동일한 배포 순서를 유지합니다. 하지만 롤포워드 및 롤백 순서는 다를 수 있습니다.

예를 들어, 일반적으로 직렬화 컨텍스트에서 특정 순서를 따릅니다. 즉, 롤포워드 중에는 읽는 쪽은 쓰는 쪽보다 먼저 진행되지만, 롤백 중에는 쓰는 쪽이 읽는 쪽보다 우선합니다. 테스트 및 프로덕션 환경에서는 일반적으로 적절한 순서에 따릅니다.

테스트 환경 설정이 프로덕션 환경과 비슷하면 가능한 한, 프로덕션 트래픽을 비슷하게 시뮬레이션합니다. 예를 들어 빠르게 연속해서 여러 레코드나 메시지를 생성하고 읽습니다. 모든 API를 지속적으로 실행합니다. 그러면 3개의 스테이지로 환경을 진행하며, 각 스테이지를 합리적인 지속 시간 동안 유지하며 잠재적 버그를 식별합니다. 이 지속 시간은 모든 API, 백엔드 워크플로 및 배치 작업이 한 번 이상 실행될 수 있을 정도로 충분해야 합니다.
첫 번째로, 플릿 절반에 변경 사항을 배포하여 소프트웨어 버전의 공존을 확인합니다. 두 번째로, 배포를 완료합니다. 세 번째로, 배포 롤백을 시작하고 모든 서버가 이전 소프트웨어를 실행할 때까지 같은 단계를 수행합니다. 이 단계 도중에 오류가 예상치 못한 동작이 없으면 테스트에 성공했다고 간주합니다.

결론

고객에 대한 서비스 중단 없이 배포를 롤백할 수 있도록 보장하는 것은 서비스 안정성의 핵심입니다. 명시적인 롤백 안전 테스트를 수행하면 오류가 잦은 수동 분석에 의존하지 않아도 됩니다. 변경이 롤백에 안전하지 않음을 발견하면 일반적으로 두 가지 변경으로 구분하고, 각각이 롤포워드와 롤백에 안전하도록 합니다.