다시 보내도 괜찮은 시스템

재시도, 멱등성, 그리고 정확히 한 번의 환상

ThakiCloud (Hyojung Han)

다시 보내도 괜찮은 시스템

재시도, 멱등성, 그리고 정확히 한 번의 환상

ThakiCloud (Hyojung Han)

Contents

1 정확히 한 번이라는 환상

결제 서버에 요청을 보냈는데 응답이 오지 않는다. 30초를 기다리다 타임아웃이 발생했다. 이제 어떻게 해야 하는가.

이 질문에 즉답할 수 있다면 이 책의 나머지는 확인용으로만 읽어도 된다. 대부분의 팀은 즉답하지 못한다. 다시 보내면 두 번 결제될까 두려운가. 보내지 않으면 고객 돈이 빠져나갔는데 주문이 없는 상태가 될까 두려운가. 그래서 보통은 사람이 개입한다. 새벽에 깨어난 누군가가 결제사 관리자 화면을 열고 눈으로 대조한다.

이 장은 그 두려움의 정체를 밝히는 데 쓴다. 결론부터 적자면, 타임아웃은 실패가 아니다. 타임아웃은 정보가 없다는 뜻이다. 네트워크로 연결된 두 컴퓨터 사이에서 정보 부재는 원리적으로 제거할 수 없는 성질이다.

1.1 응답이 사라진 요청은 무슨 일이 일어났나

요청 하나를 보내고 응답을 받지 못했을 때, 서버 쪽에서 가능한 상태는 최소 네 가지다.

- 요청이 서버에 도착하지 못했다. 아무 일도 일어나지 않았다.
- 요청이 도착했고 서버가 처리를 시작했으나 도중에 죽었다. 절반만 일어났다.
- 요청이 도착했고 처리가 완전히 끝났으나 응답이 돌아오는 길에 사라졌다. 전부 일어났다.
- 요청이 도착했고 서버는 지금도 처리 중이다. 곧 일어날 예정이다.

클라이언트가 보는 화면은 네 경우 모두 동일하다. 아무것도 오지 않았다는 사실 하나뿐이다. 여기서 중요한 것은 정보량이 정확히 0이라는 점이다. 재시도 로직을 짤 때 사람들이 흔히 하는 실수가 이 지점에서 나온다.

타임아웃을 실패로 간주하고 예외 경로를 태우는 코드는 사실 위 네 가지 중 하나를 골라 찍은 것이다. 그리고 세 번째 경우, 즉 처리는 끝났는데 응답만 사라진 경우에 그 추측은 틀린다.

여기에 더 불편한 사실이 하나 있다. 클라이언트가 확인 요청을 한 번 더 보내서 상태를 물어보면 되지 않느냐는 생각은 문제를 한 단계 미룰 뿐이다. 확인 요청도 같은 네트워크를 지나가고, 같은 방식으로 응답이 사라질 수 있다. 확인의 확인을 보내도 마찬가지다. 이것이 두 장군 문제라고 불리는 고전적 결과다. 신뢰할 수 없는 통신로 위에서는 유한한 횟수의 메시지 교환으로 양쪽이 동시에 확신에 도달할 수 없다.

실무적으로 정리하면 이렇다. 확신은 살 수 없다. 살 수 있는 것은 안전뿐이다. 안전은 받는 쪽에서 만들어진다.

1.2 두 개의 선택지, 그리고 없는 세 번째

메시지 전달 보장을 이야기할 때 세 가지 이름이 등장한다. 최대 한 번, 최소 한 번, 정확히 한 번이다. 이 중에서 실제로 구현 가능한 정책은 앞의 둘뿐이다.

정책	규칙	대가
최대 한 번	재시도 안 함	유실 발생
최소 한 번	될 때까지 재시도	중복 발생

최대 한 번은 요청을 보내고 응답이 없으면 포기한다. 중복은 절대 생기지 않지만 유실이 생긴다. 최소 한 번은 성공 응답을 받을 때까지 계속 보낸다. 유실은 없지만 중복이 생긴다. 이 둘 사이에 중간 지점은 없다. 응답을 못 받은 상태에서 재시도를 하거나 안 하거나, 딱 두 갈래뿐이기 때문이다.

그렇다면 카프카 같은 시스템이 광고하는 정확히 한 번 처리는 무엇인가. 전달 보장이 아니라 처리 효과의 보장이다. 내부적으로는 메시지를 여러 번

전달하되, 소비 결과를 트랜잭션으로 묶어 최종 상태에 한 번만 반영되게 만든다. 즉 전달은 여전히 최소 한 번이고 중복이 최종 결과에 드러나지 않도록 소비자 쪽에서 걸러낸다. 이름이 오해를 부르는데, 실체는 효과의 유일성이다.

그리고 유일성은 시스템 경계 안에서만 성립한다. 카프카 소비자가 처리 결과를 외부 결제 API에 보내는 순간, 호출은 다시 최소 한 번의 세계로 나간다. 정확히 한 번의 보장은 경계를 넘지 못한다. 이 점을 놓치면 브로커를 바꾸면 문제가 사라진다는 잘못된 기대를 갖게 된다.

1.3 그래서 우리가 실제로 만드는 것

현실의 설계 목표는 이렇게 다시 쓰인다. 전달은 최소 한 번으로 두고 같은 요청이 여러 번 도착해도 관측 가능한 결과가 한 번 실행한 것과 같도록 수신자를 만든다. 이것이 멱등성이다.

수학에서 멱등은 함수를 두 번 적용해도 한 번 적용한 것과 같다는 성질을 말한다. 시스템에서는 조금 더 실용적으로 정의한다. 같은 요청을 n번 처리했을 때 최종 상태와 외부에서 관측되는 부수효과가 1번 처리한 경우와 구별되지 않으면 그 연산은 멱등이다.

주의할 점은 여기서 관측되는 부수효과까지 포함한다는 것이다. 잔액이 한 번만 차감되었더라도 고객에게 문자가 세 통 갔다면 시스템은 멱등하지 않다. 실무에서 멱등성 사고가 나는 자리는 잔액 계산보다 알림, 웹훅, 로그 기반 정산 같은 주변부인 경우가 훨씬 많다.

멱등성은 세 가지 방식으로 얻을 수 있다.

첫째는 연산 자체를 자연스럽게 멱등으로 만드는 방법이다. 값을 더하는 대신 값을 지정하면 된다. 잔액을 100 빼는 연산은 반복하면 틀리지만, 잔액을 900으로 설정하는 연산은 몇 번을 해도 같다. 상태 갱신 계열 작업 상당수는 이 형태로 다시 쓸 수 있고, 이때는 별도 인프라가 전혀 필요 없다.

가능하면 이 방법을 먼저 검토하는 편이 좋다.

둘째는 중복 제거 저장소를 두는 방법이다. 요청마다 고유한 키를 붙이고 이미 처리한 키인지 확인한 뒤 처리한다. 자연스럽게 멱등으로 만들 수 없는 연산, 특히 돈이 오가거나 외부 시스템을 호출하는 작업에는 이 방법이 필요하다. 2장 전체가 여기에 대한 이야기다.

셋째는 상태 전이를 단조롭게 설계하는 방법이다. 상태가 앞으로만 가고 뒤로 가지 않게 만들면, 이미 지나간 전이를 다시 요청해도 아무 일도 일어나지 않는다. 3장에서 다룬다.

1.4 설계 전에 숫자로 답할 두 가지

모든 요청 경로에 멱등 키 저장소를 붙이는 것은 과잉이다. 어디에 얼마나 투자할지 정하려면 두 값을 먼저 알아야 한다.

첫째, 중복 한 번의 비용은 얼마인가. 결제라면 고객 돈이 한 번 더 빠져나가고 환불 처리와 고객 응대가 따라붙는다. 조회 API라면 서버 자원이 조금 더 쓰이고 끝난다. 이메일 발송이라면 신뢰도가 깎이지만 복구는 가능하다.

둘째, 유실 한 번의 비용은 얼마인가. 주문 이벤트가 사라지면 배송이 안 나가고 매출이 증발한다. 지표 수집 이벤트 하나가 빠지면 그래프가 아주 조금 흔들리고 만다.

이 두 값을 비교하면 정책이 자동으로 정해진다.

- 중복이 비싸고 유실도 비싸다면 최소 한 번 전달에 멱등 수신자를 붙인다. 결제, 주문, 잔액이 여기 들어간다. 투자할 가치가 확실한 구간이다.
- 중복이 싸고 유실이 비싸면 그냥 최소 한 번으로 두고 중복을 감수한다. 캐시 무효화나 재색인 요청이 대표적이다.

- 중복이 비싸고 유실이 싸면 최대 한 번을 고른다. 실시간성만 중요한 일회성 알림이 여기 해당한다.
- 둘 다 싸면 아무거나 고르고 이 문제에 시간을 더 쓰지 않는다.

여기서 자주 나오는 반론이 있다. 중복이 아주 드물게만 발생한다면 굳이 설계할 필요가 있느냐는 것이다. 확률이 낮다는 관찰은 대개 정상 상태에서 잰 값이다. 중복은 장애 중에 몰려서 발생한다. 서버가 느려지면 타임아웃이 늘고 타임아웃이 늘면 재시도가 늘고 재시도가 늘면 서버가 더 느려진다. 즉 중복률은 시스템이 가장 힘들 때 급등한다. 평시 통계로 결정을 내리면 딱 잘못될 때 잘못된다.

1.5 중복은 재시도에서만 오지 않는다

재시도를 통제하면 중복이 사라진다고 믿는 팀이 있다. 실제 장애 보고서를 모아 보면 중복의 출처는 훨씬 다양하다.

사용자가 결제 버튼을 두 번 누른다. 화면이 멈춘 것처럼 보이면 사람은 반드시 다시 누른다. 프런트엔드에서 버튼을 잠그는 처리는 새로고침 한 번으로 무력화된다.

로드밸런서나 프록시가 자체적으로 재시도한다. 게이트웨이 설정에 재시도가 켜져 있는 줄 모르고 애플리케이션에서도 재시도를 걸면 요청 수가 곱해진다. 4장에서 다룰 재시도 증폭의 씨앗이 여기서 뿌려진다.

메시지 브로커가 소비자의 확인 응답을 못 받아 같은 메시지를 다시 넣는다. 소비자는 처리를 끝냈는데 확인만 못 보냈을 수 있고, 이때 브로커는 그 차이를 알 방법이 없다.

배치 잡이 중간에 죽어서 처음부터 다시 돈다. 이미 처리한 앞부분을 또 처리한다.

운영자가 실패로 보이는 작업을 손으로 재실행한다. 이것도 중복이다.

오히려 사람이 만든 중복이 자동 재시도보다 잡기 어렵다. 로그에 재시도 표시가 남지 않기 때문이다.

이 목록의 공통점은 하나다. 어느 것도 애플리케이션 재시도 코드를 고쳐서 막을 수 없다. 그래서 방어선은 결과를 확정하는 쪽에 세워야 한다. 수신자가 안전하면 위 다섯 가지 출처를 한꺼번에 처리한다. 발신자를 고치면 그중 하나만 고친다.

1.6 관측되지 않는 중복이 더 위험하다

중복이 눈에 보이면 그나마 낫다. 문제는 조용히 지나가는 중복이다. 잔액이 두 번 깎였는데 두 건이 다른 시각에 기록되어 있으면 정산 담당자는 두 개의 정상 거래로 읽는다. 재고가 두 번 차감되었는데 그 수량이 애초에 넉넉했다면 아무도 모른다. 몇 달 뒤 재고 실사에서 숫자가 안 맞는 것으로 드러난다.

그래서 멱등성 설계에는 관측 장치가 함께 따라와야 한다. 중복 요청을 걸러냈을 때 조용히 성공만 돌려주지 말고 걸러냈다는 사실을 지표로 남겨야 한다. 이 지표는 두 가지를 알려준다. 하나는 재시도 정책이 실제로 얼마나 자주 발동하는가이고, 다른 하나는 어느 경로가 불안정한가이다. 중복 차단 건수가 갑자기 늘면 그 뒤에 반드시 원인이 있다.

1.7 이 책이 다루는 범위

지금까지의 내용을 한 문장으로 압축하면 이렇다. 재시도를 없애려고 하지 말고 재시도해도 아무 일도 일어나지 않는 상태를 만들자.

남은 세 장은 그 상태를 만드는 구체적 방법으로 채운다. 2장은 멱등 키를 어떻게 정의하고 어디에 저장하며 동시 요청에서 어떻게 무너지지 않게 하는지를 다룬다. 3장은 여러 시스템에 걸친 부분 실패, 되돌릴 수 없는 작업,

보상 트랜잭션을 다룬다. 4장은 재시도 자체를 어떻게 통제해서 장애를 키우지 않는지를 다룬다.

한 가지만 미리 말해 두면, 이 순서를 건너뛰지 않는 편이 좋다. 재시도 정책부터 손대는 팀이 많은데, 수신자가 멱등하지 않은 상태에서 재시도 횟수를 늘리는 것은 사고 규모를 키우는 일이다. 안전을 먼저 만들고 나서 재시도를 늘려야 한다.

2 멱등 키 설계와 중복 제거 저장소

같은 요청이 두 번 와도 한 번만 처리하려면, 먼저 두 요청이 동일한지 판정할 수 있어야 한다. 이 기준이 멱등 키다.

개념은 단순하지만 현장에서 문제가 되는 지점은 정해져 있다. 키를 잘못된 곳에서 만들고 범위를 잘못 잡는다. 저장을 처리보다 나중에 하고 동시 요청은 고려하지 않는다. 이 장은 네 가지 문제를 순서대로 해결한다.

2.1 키는 클라이언트가 만든다

가장 먼저 정할 것은 키 생성 주체다. 답은 요청을 보내는 쪽이다. 예외는 거의 없다.

1장에서 살펴본 내용과 같이, 서버가 요청을 받은 뒤에 키를 생성하면 재시도로 들어온 두 번째 요청은 새 키를 받는다. 서버 입장에서 두 요청은 서로 다른 요청이 되며 중복 제거는 성립하지 않는다. 두 요청이 같은지 아는 유일한 주체는 요청을 만든 쪽뿐이다.

계약은 다음과 같다. 클라이언트는 논리적으로 하나인 작업에 키를 하나 만들어 붙이고, 재시도할 때 반드시 같은 키를 다시 보낸다. 서버는 이 키를 기준으로 판정한다.

자주 발생하는 실수를 짚어 두자.

재시도할 때마다 새 키를 만드는 클라이언트가 의외로 많다. 헤더에 무작위 값을 채우는 코드를 요청 생성 함수 안에 넣으면, 재시도 루프가 그 함수를 다시 부르면서 키가 새로 생긴다. 키 생성은 재시도 루프 바깥에서 한 번만 일어나야 한다. 이 구분이 코드에 드러나 있지 않으면 언젠가 누군가 잘못 옮겨 놓는다.

요청 본문을 해시해서 키로 쓰는 방식도 위험하다. 편리해 보이지만 의도적으로 같은 내용을 두 번 보내는 정상 요청까지 막아 버린다. 같은 사람에게 같은 금액을 두 번 송금하는 일은 얼마든지 정상이다. 내용이 같다는 것과 같은 작업이라는 것은 다른 이야기다.

시각을 키에 섞는 방식도 깨진다. 재시도 시각은 원본과 다르기 때문이다.

권장은 단순하다. UUID 같은 무작위 식별자를 작업 시작 시점에 한 번 생성하고, 작업이 끝날 때까지 유지한다. 사용자 행위와 일대일로 대응하는 자연 키가 이미 있다면 그것을 써도 된다. 주문 번호가 대표적이다.

2.2 범위와 수명을 정하는 네 가지 질문

키만 정하면 끝이 아니다. 키가 어느 공간에서 유일한지, 언제까지 기억할지를 함께 정해야 한다. 네 가지 질문으로 정리된다.

첫째, 키의 유일성 범위는 어디까지인가. 전역으로 유일하게 둘 수도 있고 테넌트별 또는 엔드포인트별로 나눌 수도 있다. 실무에서는 테넌트와 엔드포인트를 키에 함께 묶는 방식이 안전하다. 어떤 고객사가 우연히 다른 고객사와 같은 키를 보내도 서로 간섭하지 않고, 같은 키로 결제 생성과 결제 취소를 부르는 사고도 막힌다.

둘째, 얼마나 오래 기억할 것인가. 영원히 보관할 수는 없다. 키 저장소는 계속 자라는 데이터이며 어느 시점에는 지워야 한다. 기준은 하나다. 이 요청을 재시도할 가능성이 있는 최대 기간보다 길게 잡는다. 클라이언트 재시도가 최대 10분이고 운영자 수동 재실행이 하루 안에 일어난다면, 보관 기간은 최소 며칠이다. 지나치게 짧게 잡으면 키가 만료된 뒤 들어온 재시도가 새 요청으로 처리된다. 이는 중복 제거 장치가 있는데도 중복이 나는, 가장 찾기 어려운 형태의 사고다.

셋째, 같은 키에 다른 내용이 오면 어떻게 하는가. 클라이언트 버그나 키 재사용으로 충분히 생긴다. 첫 요청의 본문을 해시해서 함께 저장해 두고

같은 키인데 지문이 다르면 성공을 돌려주지 말고 명시적인 충돌 오류를 반환해야 한다. 조용히 첫 번째 결과를 돌려주면 클라이언트는 두 번째 요청이 처리된 줄 안다. 이 침묵이 나중에 정산 불일치로 돌아온다.

넷째, 실패한 요청의 키는 재사용 가능한가. 정책을 명시해야 한다. 일시적 오류로 실패했다면 같은 키로 다시 시도할 수 있어야 유용하다. 반대로 영구적 거절이라면 같은 키를 다시 받아도 같은 거절을 돌려주는 편이 일관적이다. 애매하게 두면 클라이언트마다 다르게 해석한다.

2.3 진행 중 상태가 없으면 무너진다

이제 저장소 이야기다. 가장 흔한 구현은 이렇게 생겼다. 요청을 받으면 키가 있는지 조회하고 없으면 처리한 다음 키를 저장한다.

이 코드는 순차 요청에서는 잘 돈다. 그러나 동시 요청에서 반드시 깨진다.

같은 키를 가진 두 요청이 거의 동시에 들어오면, 둘 다 조회 단계에서 없음을 확인한다. 둘 다 처리로 진입한다. 둘 다 결제한다. 조회와 저장 사이에 열린 틈이 그대로 사고가 된다. 이 틈은 재시도가 몰릴 때, 즉 클라이언트가 타임아웃 직후 곧바로 다시 보낼 때 가장 잘 벌어진다. 하필 가장 필요한 순간에 뚫린다.

해법은 상태를 두 개가 아니라 세 개로 만드는 것이다.

상태	의미	응답
진행 중	처리 시작함	충돌 또는 대기
완료	결과 확정됨	저장된 결과
실패	재시도 가능	오류

핵심은 처리를 시작하기 전에 진행 중 상태를 먼저 기록하는 것이다. 이 기록은 원자적 조건부 삽입이어야 한다. 관계형 데이터베이스라면 키에

유일 제약을 걸고 삽입을 시도한다. 성공하면 이 요청이 주인이고 유일 제약 위반이 나면 다른 요청이 이미 주인이다. 레디스라면 값 설정을 존재하지 않을 때만 수행하는 명령 하나로 같은 효과를 얻는다.

이렇게 하면 조회와 저장이 하나의 원자적 연산으로 합쳐지고 틈이 사라진다. 두 번째 요청은 삽입에 실패하면서 자기가 중복이라는 사실을 즉시 알게 된다.

진행 중 상태에 걸린 요청에 무엇을 돌려줄지도 정해야 한다. 두 가지 방식이 있다. 곧바로 충돌 응답을 주고 클라이언트가 잠시 뒤 다시 물어보게 하는 방식은 단순하고 서버 자원을 아낀다. 짧게 기다렸다가 결과가 확정되면 그것을 돌려주는 방식은 클라이언트 구현이 편해진다. 어느 쪽이든 무한정 기다리게 두면 안 된다. 진행 중 상태에는 반드시 만료 시각이 있어야 하고 그 시각을 넘긴 항목은 실패로 간주해 다음 요청이 주인이 될 수 있게 열어 줘야 한다. 처리 도중 서버가 죽으면 진행 중 표시만 남고 아무도 그것을 지우지 못하기 때문이다. 이 만료 장치가 없으면 키는 영원히 잠긴다.

2.4 결과를 함께 저장한다

중복을 걸러내는 것만으로는 부족하다. 두 번째 요청에 무엇을 돌려줄지가 남는다.

성공했다는 사실만 돌려주면 클라이언트는 결제 번호나 생성된 자원 식별자를 알 수 없다. 원래 응답 본문을 키와 함께 저장해 두고 중복 요청에는 저장된 응답을 그대로 반환한다. 클라이언트 입장에서는 첫 번째 요청이 성공한 것과 완전히 같은 경험이 된다. 응답 상태 코드까지 같이 저장해 두면 더 깔끔하다.

순서가 중요하다. 결과 저장과 실제 작업 반영이 같은 트랜잭션 안에 있어야 한다. 작업을 반영하고 커밋한 뒤에 별도로 키 저장소를 갱신하는 구조는, 그 사이에 죽으면 작업은 반영되었는데 완료 표시가 없는 상태를 만든다.

다음 재시도가 다시 처리한다. 애써 만든 장치가 그 순간 무력해진다.

같은 데이터베이스를 쓴다면 하나의 트랜잭션으로 묶는 것이 가장 확실하다. 레디스처럼 별도 저장소를 쓴다면 구조적 취약점을 인정하고, 실제 작업 쪽에도 방어선을 하나 더 둬야 한다. 다음 장의 단조로운 상태 전이가 그 두 번째 방어선 역할을 한다.

2.5 키가 지켜 주지 못하는 구간

멱등 키를 붙여 놓고도 중복이 나는 사례가 있다. 원인은 대개 키가 덮는 구간과 실제 부수효과가 일어나는 구간이 어긋난 데 있다.

예를 들어 결제 API 앞단에 키 검사를 붙였는데, API가 내부적으로 결제사, 쿠폰 서비스, 알림 서비스를 차례로 호출한다고 하자. 첫 요청이 결제사 호출까지 성공하고 쿠폰 서비스에서 죽으면, 진행 중 표시는 만료되어 풀리고 재시도가 들어온다. 두 번째 시도는 처음부터 다시 시작해 결제사를 다시 호출한다. 바깥 경계에는 키가 있었지만 결제사 호출에는 없기 때문이다.

규칙은 이렇게 잡는다. 멱등 키는 되돌릴 수 없는 부수효과가 일어나는 지점마다 있어야 한다. 하나의 요청이 되돌릴 수 없는 호출을 세 번 한다면 방어선도 세 개다. 바깥 경계의 키는 전체 요청을 한 번으로 만들어 주지만 안쪽이 도중에 끊기는 경우까지 책임지지는 못한다.

다행히 안쪽 방어선은 대개 이미 있다. 결제사나 외부 API 대부분이 자체 멱등 키 헤더를 받는다. 우리가 할 일은 그 헤더에 무엇을 넣을지 정하는 것이다. 여기서도 원칙은 같다. 재시도해도 변하지 않는 값이어야 한다. 우리 쪽 작업 식별자를 그대로 파생시켜 쓰면 자연스럽게 만족한다. 예를 들어 우리 결제 시도 번호에 고정 접미사를 붙여 만든다. 이렇게 하면 우리가 몇 번을 재시도하든 외부에 나가는 키는 항상 같다.

2.6 재시도와 새 시도를 구분하는 법

미묘한 설계 선택이 하나 생긴다. 사용자가 결제에 실패한 뒤 다시 시도하는 것은 재시도인가 새 시도인가.

시스템 관점에서는 새 시도로 보는 편이 맞다. 사용자가 카드 정보를 고쳤을 수도 있고 실패 원인이 해소되었을 수도 있다. 같은 키로 묶으면 첫 번째 실패 결과를 계속 돌려주게 된다.

계층을 나눈다. 사용자의 한 번의 시도가 하나의 결제 시도 레코드이고 레코드마다 멱등 키가 하나씩 붙는다. 네트워크 재시도는 같은 레코드 안에서 같은 키를 재사용하고 사용자가 다시 누르는 것은 새 레코드와 새 키를 만든다. 이 구분이 코드에 명시되어 있어야 한다. 명시하지 않으면 두 개념이 섞이고, 어떤 팀은 사용자 재시도를 막고 어떤 팀은 네트워크 재시도를 중복으로 통과시킨다.

주문 단위에서 진짜 중복을 막고 싶다면 그것은 멱등 키의 일이 아니라 비즈니스 제약의 일이다. 같은 주문에 결제 완료 레코드가 둘 이상 생기지 않도록 데이터 모델에 유일 제약을 거는 편이 확실하다. 두 장치는 목적이 다르고 둘 다 필요하다.

2.7 저장소를 고르는 기준

어디에 저장할지는 세 가지로 갈린다.

작업 데이터와 같은 관계형 데이터베이스에 두면 트랜잭션으로 묶을 수 있어 가장 안전하다. 대신 쓰기 부하가 데이터베이스에 더해진다. 결제나 주문처럼 정확성이 최우선인 경로에는 이쪽을 권한다.

레디스 같은 인메모리 저장소는 빠르고 만료 처리가 편하다. 대신 작업과 원자적으로 묶이지 않고 구성에 따라 데이터가 사라질 수 있다. 조회성 API나 알림 발송처럼 중복 비용이 크지 않은 경로에 맞는다.

메시지 큐 소비자라면 처리 결과를 기록하는 테이블 자체가 중복 제거 저장소 역할을 겸할 수 있다. 메시지 식별자를 유일 키로 두고 결과와 함께 삽입하면 별도 인프라 없이 끝난다. 이미 있는 테이블을 쓰는 것이 새 저장소를 도입하는 것보다 거의 항상 낫다.

마지막으로 한 가지. 중복 제거 저장소는 조용히 커진다. 만료 정책을 처음부터 넣어 두고 저장소 크기와 중복 차단 건수를 지표로 뽑아 두자. 이 두 숫자가 없으면 문제가 생겼을 때 언제부터 잘못되었는지 알 수 없다.

3 단조로운 상태와 부분 실패

2장의 중복 제거 저장소는 강력하지만 인프라를 하나 더 얹는 일이다. 저장소가 흔들리면 방어선도 함께 흔들린다.

더 값싸고 더 튼튼한 두 번째 방어선이 있다. 데이터 모델 자체를 재시도에 무감각하게 만드는 방법이다. 이 장은 그 기법과, 하나의 데이터베이스로 감쌀 수 없는 부분 실패를 다룬다.

3.1 앞으로만 가는 상태

주문 상태가 생성, 결제완료, 배송중, 배송완료 순으로 흐른다고 하자. 이때 규칙을 하나만 추가한다. 상태는 정해진 순서로만 전진하고 절대 후퇴하지 않는다.

이 규칙이 있으면 상태 갱신 코드는 이렇게 쓰인다. 현재 상태가 결제완료보다 앞일 때만 결제완료로 바꾼다. 조건은 갱신문 안에 들어간다.

같은 요청이 두 번 들어오면 어떻게 되나. 첫 번째는 조건을 만족해 상태를 바꾼다. 두 번째는 이미 결제완료이므로 조건을 만족하지 못하고 아무 행도 바꾸지 못한다. 별도 키 저장소 없이 재시도가 안전해졌다.

조건은 조회로 확인하지 않고 갱신문 자체에 넣어야 한다는 점이 중요하다. 조회한 뒤 판단하고 갱신하는 세 단계로 나누면 2장에서 본 그 틈이 그대로 생긴다. 데이터베이스가 조건 판정과 갱신을 하나로 처리하게 맡겨야 한다. 그리고 갱신된 행 수를 반드시 확인해야 한다. 0이면 이미 지나간 전이라는 뜻이고, 이것은 오류가 아니라 정상적인 중복 처리 결과다. 이 값을 무시하는 코드가 의외로 많다.

단조성은 상태 문자열에만 적용되는 개념이 아니다. 몇 가지 형태로

나타난다.

- 순서가 있는 상태 값에서는 뒤로 못 가게 막는다.
- 한 번 참이 되면 거짓이 되지 않는 표시를 쓴다. 발송됨, 승인됨 같은 값이 여기 해당한다.
- 시각이나 버전 번호가 더 큰 갱신만 받아들인다. 늦게 도착한 옛 메시지를 자동으로 걸러 준다.
- 집합에 원소를 추가하기만 하고 빼지 않는다. 같은 원소를 여러 번 넣어도 결과가 같다.

마지막 형태는 특히 유용하다. 처리한 항목 식별자를 집합에 넣어 가며 진행하는 배치 잡은 중간에 죽고 다시 시작해도 이미 처리한 항목을 건너뛴다.

3.2 값을 더하지 말고 지정하라

숫자를 다룰 때 재시도에 취약해지는 전형적 형태가 상대 갱신이다. 재고를 1 줄이는 연산, 포인트를 100 더하는 연산이 그렇다. 두 번 실행되면 두 번 반영된다.

가능하면 절대값 지정으로 바꾼다. 재고를 47로 설정하고 포인트를 3200으로 설정한다. 몇 번을 실행해도 결과가 같다. 물론 이러려면 계산 시점의 값을 알아야 하고, 그 사이 다른 요청이 값을 바꿨을 수 있다. 그래서 버전 번호를 함께 조건으로 걸어 준다. 읽을 때 본 버전과 지금 버전이 같을 때만 갱신한다. 다르면 다시 읽고 다시 계산한다.

상대 갱신을 도저히 피할 수 없는 경우도 있다. 동시 차감이 잦은 재고 같은 자원은 절대값 지정이 오히려 경합을 키운다. 이때는 상대 갱신을 유지하되 그 연산에는 반드시 2장의 키 방어선을 붙인다. 방법을 고르는 기준은 취향이 아니라 경합 정도다.

3.3 이중 쓰기는 원자적이지 않다

여기까지는 데이터베이스 하나 안의 이야기였다. 현실은 대개 그렇지 않다.

주문을 저장하고 나서 메시지 큐에 이벤트를 발행한다고 하자. 코드는 두 줄이지만 그 사이에는 실패할 수 있는 틈이 있다. 저장은 됐는데 발행 전에 죽으면 이벤트가 영원히 나가지 않는다. 발행은 됐는데 트랜잭션이 롤백되면 없는 주문에 대한 이벤트가 돌아다닌다.

이 문제를 재시도로 풀려는 시도는 대개 실패한다. 두 시스템에 걸친 원자성은 재시도로 만들어지지 않기 때문이다. 표준적인 해법은 쓰기 대상을 하나로 줄이는 것이다.

아웃박스 패턴이 그 방법이다. 이벤트를 큐에 직접 넣지 않고 주문과 같은 데이터베이스의 아웃박스 테이블에 넣는다. 주문 저장과 이벤트 기록이 하나의 트랜잭션으로 커밋된다. 별도의 발행기가 아웃박스를 읽어 큐로 보내고, 성공하면 발행됨으로 표시한다.

발행기가 이벤트를 보내고 표시하기 전에 죽으면 같은 이벤트가 두 번 나간다. 그래도 괜찮다. 이 구조는 원자성을 최소 한 번으로 낮추고, 중복은 소비자 쪽 멱등성으로 흡수한다는 계약이기 때문이다. 아웃박스 행의 식별자를 이벤트에 실어 보내면 소비자가 그것을 중복 제거 키로 쓸 수 있다. 1장에서 말한 구조가 여기서 그대로 반복된다. 전달은 최소 한 번, 효과는 한 번이다.

3.4 되돌릴 수 없는 것들

트랜잭션의 세계에서는 실패하면 롤백하면 된다. 바깥세상에는 롤백이 없다.

돈은 이미 나갔고 메일은 이미 도착했고 물건은 이미 출고됐다. 이런 작업이 섞인 흐름은 원자적으로 만들 수 없다. 대신 앞으로 나아가면서 되돌린다.

이것이 보상 트랜잭션이다.

세 단계짜리 흐름을 생각해 보자. 결제하고 재고를 잡고 배송을 요청한다. 배송 요청이 실패하면 앞의 두 단계를 보상해야 한다. 재고를 풀고 결제를 환불한다. 환불은 결제를 없던 일로 만들지 않는다. 결제 기록과 환불 기록이 둘 다 남고 순 효과가 0이 된다. 회계 장부와 같은 방식이다.

보상 설계에서 자주 빠뜨리는 것이 세 가지 있다.

첫째, 보상 자체도 멱등해야 한다. 환불 요청이 타임아웃 나면 환불도 재시도한다. 두 번 환불되면 사고 규모가 두 배가 된다. 보상 작업에도 멱등 키를 붙여야 한다. 원본 작업 식별자에서 파생시키면 자연스럽다.

둘째, 보상은 실패할 수 있다. 환불 API가 죽어 있으면 보상이 안 끝난다. 이때 흐름을 그냥 실패로 닫으면 돈이 붕 뜬 상태로 남는다. 보상 대기 상태를 명시적으로 두고 재시도하고, 일정 횟수를 넘기면 사람에게 알려야 한다. 보상 실패는 조용히 지나가면 안 되는 몇 안 되는 사건이다.

셋째, 보상 순서는 역순이다. 그리고 각 단계는 자기가 실제로 실행되었는지 확인한 뒤 보상해야 한다. 재고를 잡지 못한 상태에서 재고 해제를 부르면 없는 수량을 되돌린다. 각 단계의 실행 여부를 기록해 두고 그 기록을 보고 보상 대상을 정한다.

3.5 사가를 언제 쓰고 언제 피하나

보상 트랜잭션을 여러 단계로 엮은 흐름을 사가라고 부른다. 이름이 붙어 있다 보니 분산 시스템이면 일단 사가를 도입해야 한다고 생각하기 쉬운데, 실제로는 도입을 미루는 편이 나은 경우가 더 많다.

사가는 원자성을 포기하는 대신 가용성을 얻는 거래다. 중간 상태가 바깥에 노출된다는 뜻이기도 하다. 결제는 됐는데 배송이 아직 잡히지 않은 순간에 고객이 주문 조회를 하면 무엇을 보여줄 것인가. 이 질문에 답이 없으면

사가는 아직 이르다.

판단 기준을 세 가지로 잡는다.

한 데이터베이스 안에서 끝나는가. 끝난다면 그냥 트랜잭션 하나로 처리한다. 마이크로서비스라서 나눠야 한다는 말은 기술적 이유가 아니라 조직적 이유인 경우가 많다.

되돌릴 수 없는 단계가 몇 개인가. 하나뿐이라면 그 단계를 흐름의 맨 끝으로 옮기는 것만으로 대부분 해결된다. 앞 단계가 다 성공한 뒤에 마지막으로 실행하면 보상할 일이 생기지 않는다. 이 재배치는 사가를 만드는 것보다 훨씬 싸다.

중간 상태를 사용자에게 설명할 수 있는가. 처리 중이라는 상태를 화면에 그릴 수 있고 고객 지원팀이 그 의미를 안다면 사가를 써도 된다. 그렇지 않으면 중간 상태는 문의 전화가 된다.

3.6 흐름은 명시적으로 저장한다

사가를 쓰기로 했다면 진행 상황을 반드시 어딘가에 기록해야 한다. 코드의 실행 위치가 곧 상태인 구조, 즉 함수 호출 스택에만 진행 상황이 들어 있는 구조는 프로세스가 죽는 순간 전부 사라진다.

기록해야 할 것은 세 가지다. 지금 어느 단계까지 왔는가, 각 단계가 실제로 실행되었는가, 각 단계에 어떤 외부 식별자가 발급되었는가. 마지막 항목이 특히 중요하다. 결제사가 돌려준 거래 번호를 저장해 두지 않으면 나중에 환불할 방법이 없다.

이 기록이 있으면 프로세스가 죽어도 다른 인스턴스가 이어받을 수 있다. 그리고 이어받는 동작 자체가 재시도이므로, 각 단계는 앞에서 말한 방어선을 그대로 갖추고 있어야 한다. 결국 모든 이야기가 같은 자리로 돌아온다. 안전한 단계들을 이어 붙이면 안전한 흐름이 되고, 안전하지

않은 단계는 어떤 조율 장치로도 구제되지 않는다.

3.7 부수효과를 등급으로 나눈다

모든 부수효과를 같은 강도로 방어할 필요는 없다. 세 등급으로 나누면 어디에 힘을 쓸지 명확해진다.

등급	예시	필요한 장치
되돌릴 수 없음	송금, 출고	멱등 키 필수
되돌릴 수 있음	상태 변경	단조 전이
흔적만 남음	조회, 캐시	방어 불필요

첫째 등급은 실행되면 끝이다. 여기에는 2장의 키 방어선과 3장의 상태 방어선을 모두 건다. 코드 리뷰에서 가장 오래 붙들고 봐야 할 자리다.

둘째 등급은 되돌릴 수 있으니 단조 전이만으로 충분한 경우가 많다. 여기까지 무거운 인프라를 얹으면 복잡도만 늘어난다.

셋째 등급은 반복되어도 자원만 조금 더 쓴다. 그냥 둔다.

이 분류를 팀이 공유하고 있으면 새 기능을 만들 때 논쟁이 짧아진다. 이 호출이 몇 등급이냐고 물으면 필요한 장치가 바로 나온다.

3.8 알림이 새는 자리

마지막으로 실무에서 가장 자주 새는 구멍을 하나 짚는다. 알림이다.

잔액 차감은 트랜잭션 안에 있고 멱등 키도 붙어 있는데, 결제 완료 문자 발송은 트랜잭션 커밋 뒤에 별도로 호출되는 구조가 흔하다. 재시도가 들어오면 잔액은 안전하게 걸러지는데, 걸러진 뒤에도 알림 코드가

실행되어 문자가 또 나간다. 중복 요청 경로가 성공 응답과 함께 알림까지 다시 태우기 때문이다.

고치는 방법은 두 가지다. 알림 발송을 아웃박스에 넣어 상태 전이와 함께 커밋하거나, 중복으로 판정된 요청은 저장된 응답만 돌려주고 그 아래 로직을 아예 태우지 않는 것이다. 후자가 더 단순하고 대개 충분하다.

원칙으로 정리하면 이렇다. 중복 판정 이후의 코드 경로는 순수해야 한다. 저장된 결과를 읽어 돌려주는 것 외에 아무것도 하지 않는다. 이 한 줄을 코드 리뷰 체크리스트에 넣어 두면 상당수의 사고가 사라진다.

4 재시도 정책과 폭주 차단

앞의 세 장은 재시도가 들어와도 안전한 수신자를 만드는 이야기였다. 이제 보내는 쪽으로 돌아간다.

수신자가 안전해졌다고 해서 마음껏 재시도해도 되는 것은 아니다. 잘못 설계된 재시도는 그 자체로 장애를 만든다. 정확히 말하면, 작은 장애를 큰 장애로 키우는 가장 흔한 원인이 재시도다. 이 장은 재시도를 통제하는 방법과 네 가지 실제 경로에 적용하는 점검표로 마무리한다.

4.1 재시도해도 되는 실패를 코드가 판정하게 한다

첫 번째 규칙은 모든 실패를 재시도하지 않는 것이다.

인증이 틀렸는데 다시 보내면 또 틀린다. 요청 본문 형식이 잘못됐는데 다시 보내도 똑같다. 이런 실패를 재시도하는 것은 서버 자원을 태우면서 결과를 늦추는 일일 뿐이다. 반대로 일시적 과부하나 네트워크 끊김은 재시도로 대부분 해결된다.

판정 기준은 사람의 감이 아니라 코드에 표로 들어 있어야 한다.

실패 유형	재시도	근거
연결 실패	함	도달 못 함
타임아웃	조건부	상태 모름
서버 과부하	함	일시적
인증 오류	안 함	반복해도 같음
요청 형식 오류	안 함	고쳐야 함
자원 없음	안 함	사라진 대상

타임아웃 줄이 조건부인 이유는 1장에서 설명했다. 요청이 도달했는지 알

수 없기 때문이다. 수신자가 멱등하면 재시도해도 되고 멱등하지 않으면 재시도 대신 상태 조회로 넘어가야 한다. 즉 이 칸의 답은 상대방의 성질에 달려 있다. 그래서 외부 API를 연동할 때 가장 먼저 확인할 것은 그쪽이 멱등 키를 지원하는가이다. 지원하면 타임아웃을 재시도할 수 있고 지원하지 않으면 타임아웃마다 사람이 개입해야 한다.

서버가 응답에 재시도 가능 여부를 실어 주면 클라이언트 구현이 훨씬 단순해진다. 오류 본문에 재시도 가능 표시와 권장 대기 시간을 넣는 관례를 팀 안에서 세우면, 클라이언트마다 다르게 추측하는 일이 없어진다.

4.2 백오프와 지터

재시도한다고 정했다면 언제 보낼지를 정해야 한다. 즉시 다시 보내는 방식은 최악이다. 상대가 과부하로 실패했는데 바로 또 때리면 회복할 시간을 주지 않는다.

지수 백오프가 기본이다. 대기 시간을 1초, 2초, 4초, 8초처럼 배로 늘린다. 초기 실패는 빠르게 재시도해서 순간적인 끊김을 흡수하고 계속 실패하면 간격을 벌려 상대에게 숨 쉴 틈을 준다.

여기에 지터를 반드시 더한다. 지터는 대기 시간에 무작위성을 섞는 것이다. 없으면 어떻게 되는지 그려 보면 이해가 빠르다. 서버가 1초 동안 죽었다가 살아난다. 그동안 실패한 클라이언트 1000개가 모두 정확히 1초 뒤에 재시도한다. 살아난 서버는 동시에 도착한 1000개 요청에 다시 쓰러진다. 그리고 2초 뒤에 또 같은 일이 벌어진다. 재시도가 스스로 만든 주기적 파도가 회복을 계속 방해한다.

지터를 넣으면 이 파도가 흩어진다. 구현은 단순하다. 계산된 대기 시간 구간 안에서 무작위 값을 뽑으면 된다. 배 단위로 늘린 값을 그대로 쓰지 말고 0부터 그 값 사이에서 고르는 방식이 가장 널리 쓰이고, 실제로 잘 동작한다.

한 가지 덧붙이면, 상한도 정해야 한다. 무한정 배로 늘리면 대기 시간이 몇 시간이 된다. 보통 30초에서 몇 분 사이에 천장을 두고 그 뒤로는 고정 간격으로 간다.

4.3 재시도 증폭

가장 무서운 실패 형태는 계층 곱셈이다.

클라이언트가 3번 재시도하고 그 요청을 받는 게이트웨이가 3번 재시도하고 게이트웨이 뒤의 서비스가 다시 3번 재시도한다고 하자. 각 계층은 합리적인 숫자를 골랐다. 그런데 가장 안쪽 서비스가 받는 요청 수는 27배가 된다. 계층이 하나 더 있으면 81배다.

이 상황은 부하가 없을 때는 절대 보이지 않는다. 정상 상태에서는 재시도가 거의 발생하지 않기 때문이다. 안쪽 서비스가 조금 느려지는 순간 곱셈이 켜지고 그 순간 부하가 수십 배로 뛰면서 완전히 쓰러진다. 처음의 작은 지연이 전면 장애로 번지는 전형적 경로다.

막는 방법은 세 가지다.

재시도 계층을 하나로 정하는 것이 먼저다. 보통은 사용자와 가장 가까운 바깥 계층 하나만 재시도하고 안쪽은 전부 끈다. 이 결정은 문서로 남기고 게이트웨이 설정까지 확인해야 한다. 프록시 기본 설정에 재시도가 켜져 있는 줄 모르는 경우가 정말 많다.

다음은 재시도 예산이다. 횟수 제한이 아니라 비율 제한이다. 전체 요청 대비 재시도가 차지하는 비율이 예를 들어 10퍼센트를 넘으면 그 이상은 재시도하지 않고 즉시 실패시킨다. 이렇게 하면 개별 요청이 몇 번 재시도하든 시스템 전체의 추가 부하는 상한이 정해진다. 이 장치 하나가 위 곱셈 문제 대부분을 무력화한다.

마지막으로 재시도 요청에 표시를 남긴다. 헤더에 재시도 횟수를 실어

보내면 서버가 이 요청이 몇 번째인지 알 수 있고, 지표로도 뽑을 수 있다. 재시도 비율이 갑자기 뛰는 것을 알림으로 잡으면 장애를 조기에 발견한다.

4.4 서킷 브레이커와 데드레터

상대가 확실히 죽어 있을 때는 재시도조차 낭비다. 서킷 브레이커는 실패율이 임계를 넘으면 회로를 열어 그 뒤 요청을 즉시 실패시킨다. 일정 시간이 지나면 시험 요청 하나를 보내 보고 성공하면 다시 닫는다.

이 장치의 진짜 가치는 상대를 보호하는 것보다 나를 보호하는 데 있다. 죽은 서비스를 계속 호출하면 내 스레드와 커넥션이 거기 묶이고, 결국 내 서비스도 다른 요청을 못 받는다. 회로를 빨리 열면 그 자원이 풀린다.

그리고 모든 재시도에는 끝이 있어야 한다. 재시도 예산을 다 쓴 요청은 어디로 가나. 이 질문에 답을 정해 두지 않으면 실패가 로그 한 줄로 사라진다.

메시지 큐에서는 데드레터 큐가 그 답이다. 정해진 횟수만큼 실패한 메시지를 별도 큐로 옮긴다. 여기서 두 가지를 반드시 해야 한다. 하나는 데드레터 큐의 크기를 감시하는 것이다. 만들어 놓고 아무도 안 보는 데드레터 큐는 데이터를 조용히 버리는 장치와 같다. 다른 하나는 재처리 경로를 미리 만들어 두는 것이다. 원인을 고친 뒤 메시지를 다시 넣는 도구가 없으면, 장애 복구 시점에 급하게 스크립트를 짜게 된다.

동기 요청 경로에서도 같은 원칙이 적용된다. 최종 실패한 요청을 별도 테이블에 남기고 그 테이블을 보는 사람이 있어야 한다.

4.5 네 가지 경로 점검표

지금까지의 내용을 실제 경로에 적용해 보자.

결제 API를 호출할 때는 결제 시도마다 멱등 키를 만들어 들고 다니고

타임아웃에서는 재시도 전에 상태 조회를 먼저 시도한다. 재시도 횟수는 짧게 잡고 한계를 넘으면 자동 판단을 포기하고 미확정 상태로 표시한 뒤 정산 대사에 넘긴다. 결제에서 가장 나쁜 결과는 실패가 아니라 상태를 모르는 채 성공으로 처리하는 것이다.

메시지 큐 소비자는 전달이 최소 한 번임을 전제로 짠다. 메시지 식별자를 유일 키로 하는 처리 기록 테이블을 두고, 처리 결과와 함께 한 트랜잭션으로 커밋한다. 확인 응답은 처리 완료 뒤에 보낸다. 먼저 보내면 유실이 생긴다. 데드레터 큐와 재처리 도구를 함께 준비한다.

잡 스케줄러는 같은 잡이 두 번 뜨는 상황을 정상으로 간주한다. 스케줄러 자체가 최소 한 번 보장인 경우가 많고 배포 중 인스턴스가 겹치기도 한다. 잡 실행 단위마다 실행 식별자를 두고 조건부 삽입으로 소유권을 잡는다. 그리고 잡을 처음부터 다시 돌려도 되도록 짠다. 처리한 항목을 기록하며 진행하면 자연스럽게 만족한다.

추론 API처럼 호출 한 번이 비싼 경로에서는 재시도 비용이 특히 크다. 요청 단위 멱등 키로 결과를 캐시해 두면 재시도가 같은 결과를 즉시 돌려받는다. 타임아웃 값은 실제 응답 시간 분포를 보고 정한다. 너무 짧게 잡으면 아직 계산 중인 요청을 버리고 다시 요청하게 되어 부하가 배로 든다. 그리고 스트리밍 응답은 중간에 끊겼을 때 처음부터 다시 받을지 이어받을지 정책을 정해 둬야 한다.

4.6 마무리

이 책의 주장을 한 문단으로 줄이면 이렇다.

네트워크 위에서 정확히 한 번은 살 수 없는 물건이다. 대신 우리는 두 가지를 산다. 전달은 최소 한 번으로 넉넉히 하고 효과는 수신자 쪽에서 한 번으로 좁힌다. 좁히는 도구는 세 가지다. 자연스럽게 멱등한 연산으로 다시 쓰기, 멱등 키와 중복 제거 저장소, 단조로운 상태 전이다. 그리고 그

위에 재시도를 통제하는 정책을 얹어 장애가 스스로 커지지 않게 한다.

마지막으로 실천 순서를 남긴다. 새 경로를 만들 때 이 순서로 물어보면 대개 빠뜨리지 않는다.

이 작업에 되돌릴 수 없는 부수효과가 있는가. 없으면 여기서 끝이다. 있으면 그 지점마다 멱등 키가 있는가. 상태 전이가 단조로운가. 중복으로 판정된 요청이 아무 부수효과도 일으키지 않고 저장된 결과만 돌려주는가. 재시도 정책에 백오프와 지터와 예산이 있는가. 최종 실패한 요청이 갈 곳이 정해져 있고 그곳을 보는 사람이 있는가.

여섯 개다. 새벽에 깨어나 관리자 화면을 대조하는 일을 줄이려면, 이 여섯 개를 코드 리뷰 체크리스트에 올려 두는 것으로 시작하면 된다.