2️⃣

전자 지갑

문제 이해 및 설계 범위 확정

요구사항

- 전자 지갑 간 이체
- 1,000,000TPS
- 99.99% 안정성
- 트랜잭션
- 재현성

개략적 추정

- TPS를 거론한다는 것은 배후에 트랜잭션 기반 데이터베이스를 사용한다는 뜻
- 본 설계안에서 사용할 데이터베이스 노드는 1,000TPS를 지원한다고 가정 = 1,000개의 데이터베이스 노드 필요
- 이체를 실행하려면 인출, 입금 총 두 번의 연산이 필요함 = 1백만 건의 TPS를 처리하기 위해서는 2백만 TPS를 지원해야 함 = 2,000개의 데이터베이스 노드 필요
- 하드웨어가 같다고 가정했을 때 한 노드가 초당 처리할 수 있는 트랜잭션 수가 많을 수록 필요한 총 노드 수는 줄어듦
- 본 설계안의 목표 중 하나는 단일 노드가 처리할 수 있는 트랜잭션 수를 늘리는 것

개략적 설계안 제시 및 동의 구하기

API 설계

- 한 지갑에서 다른 지갑으로 자금 이체

  [POST] /v1/wallet/balance_transfer

필드	설명	자료형
from_account	자금을 인출할 계좌	string
to_account	자금을 이체할 계좌	string
amount	이체할 금액	string
currency	통화 단위	string
transaction_id	중복 제거에 사용할 ID	uuid

```
// 응답 값
{
    "status":"success",
    "transaction_id":""
}
```

인메모리 샤딩

- 지갑 애플리케이션은 모든 사용자 계정의 잔액을 유지함
  - <사용자, 잔액> 관계를 나타내기에 좋은 자료 구조 = 해시 테이블/맵(map)/키-값 저장소
- Redis
  - Redis 노드 한 대로 100만 TPS 처리는 벅참
  - 클러스터를 구성하고 사용자 계정을 모든 노드에 균등하게 분산시킬 수 있는 파티셔닝/샤딩이 필요함
  - 키-값 데이터를 n개 파티션에 고르게 분배
    - 키의 해시 값을 계산하고 이를 파티션의 수 n으로 나누기
    - 나머지 값이 데이터를 저장할 파티션 번호

    ```
    String accountId = "A";
    Int partitionNumber = 7;
    Int myPartition = accountID.hashCode() % partitionNumbe
    ```

  - 모든 Redis 노드의 파티션 수 및 주소는 한군데 저장해 둠
    - 높은 가용성을 보장하는 걸정 정보 전문 저장소 주키퍼를 사용하면 좋음

- 이 방안의 마지막 구성 요소 = 이체 명령 처리를 담당하는 서비스 = 지갑 서비스
  - 이체 명령의 수신
  - 이체 명령의 유효성 검증
  - 명령이 유효한 것으로 확인되면 이체에 관계된 두 계정의 잔액 갱신
    - 두 계정은 서로 다른 Redis 노드에 있을 수 있음
  - 무상태 서비스
    - 수평적 규모 확장이 용이함
  - 정확성의 요구사항을 충족하지 못함
    - 지갑 서비스는 이체할 때마다 두 개의 Redis 노드를 업데이트하는데 그 두 연산이 모두 성공하리라는 보장이 없기 때문에

    ⇒ 두 업데이트 연산은 하나의 원자적 트랜잭션으로 실행되어야 함

분산 트랜잭션

데이터베이스 샤딩

서로 다른 두 개 저장소 노드를 갱신하는 연산을 원자적으로 수행하려면 어떻게 해야 할까?

⇒ 각 Redis 노드를 트랜잭션을 지원하는 관계형 데이터베이스 노드로 교체

- 클라이언트 A, B, C의 잔액 정보가 Redis 노드가 아닌 3개의 관계형 데이터베이스 노드로 분산됨
- 트랜잭션 데이터베이스를 사용해도 일부의 문제만 해결 가능
  - 서로 다른 두 데이터베이스 서버에 있는 계정 두 개를 업데이트해야 할 가능성이 높음
- 첫 번째 계정의 잔액을 갱신한 직후에 지갑 서비스가 재시작된 경우에 두 번째 계정의 잔액도 반드시 갱신되도록 하려면 어떻게 해야 할까?

분산 트랜잭션: 2단계 커밋/2PC

- 저수준 방안
  - 데이터베이스 자체에 의존하는 방향
- 두 단계로 실행됨

1. 조정자는 정상적으로 여러 데이터베이스에 읽기 및 쓰기 작업을 수행
2. 애플리케이션이 트랜잭션을 커밋하려 할 때 조정자는 모든 데이터베이스에 트랜잭션 준비를 요청함
3. 두 번째 단계에서 조정자는 모든 데이터베이스의 응답을 받아 다음 절차를 수행함
    a. 모든 데이터베이스가 '예'라고 응답하면 조정자는 모든 데이터베이스에 해당 트랜잭션 커밋을 요청함
    b. 어느 한 데이터베이스라도 '아니요'를 응답하면 조정자는 모든 데이터베이스에 트랜잭션 중단을 요청함

- 준비 단계를 실행하려면 데이터베이스 트랜잭션 실행 방식을 변경해야 함
    - 이기종 데이터베이스 사이에 2PC를 실행하려면 모든 데이터베이스가 X/OPEN XA 표준을 만족해야 함
- 다른 노드의 메시지를 기다리는 동안 락이 오랫동안 잠긴 상태로 남을 수 있어서 성능이 좋지 않음
- 조정자가 SPOF이 될 수도 있음

분산 트랜잭션: TC/C

- Try-Confirm/Cancel 두 단계로 구성된 보장 트랜잭션
    1. 조정자는 모든 데이터베이스에 트랜잭션에 필요한 자원 예약을 요청
    2. 조정자는 모든 데이터베이스로부터 회신을 받음
        a. 모든 데이터베이스가 '예'라고 응답하면 조정자는 모든 데이터베이스에 작업 확인을 요청함 ⇒ Try-Confirm
        b. 어느 한 데이터베이스라도 '아니요'를 응답하면 조정자는 모든 데이터베이스에 작업 취소를 요청함 ⇒ Try-Cancel
- 각 단계가 별도 트랜잭션임
- 사례

단계	실행연산	A	C
1	시도	잔액 변경: -$1	아무것도 하지 않음
2	확인	아무것도 하지 않음	잔액 변경: +$1
	취소	잔액 변경: +$1	아무것도 하지 않음

    - 첫 번째 단계: 시도

        - 조정자 역할을 하는 지갑 서비스가 두 개의 트랜잭션 명령을 데이터베이스로 전송
        - 조정자는 계정 A가 포함된 데이터베이스에 A의 잔액 1달러를 감소시키는 트랜잭션 시작
        - 조정자는 계정 C가 포함된 데이터베이스에는 아무 작업을 하지 않음
    - 두 번째 단계: 확정
        - 두 데이터 베이스가 모두 '예'라고 응답하면 지갑 서비스는 확정 단계를 실행
        - A의 잔액은 이미 첫 번째 단계에서 갱신됨
        - 계정 C의 잔액에 $1 추가
    - 두 번째 단계: 취소
        - 첫 번째 시도 단계가 실패
        - 시도 단계에서 이미 계정 A의 잔액은 바뀌었고 트랜잭션은 종료됨
        - 이미 종료된 트랜잭션의 효과를 되돌리려면 지갑 서비스는 또 다른 트랜잭션을 시작하여 계정 A에 1달러를 추가함
- 보상 기반 분산 트랜잭션
- 실행 취소 절차를 비지니스 로직으로 구현함 ⇒ 고수준 해법
- 특정 데이터베이스에 구애 받지 않음 = 트랜잭션을 지원하는 데이터베이스이기만 하면 TC/C는 작동함
- 애플리케이션 계층의 비지니스 로직에서 세부사항을 관리하고 분산 트랜잭션의 복잡성을 처리해야 함

2PC vs TC/C

	첫 번째 단계	두 번째 단계: 성공	두 번째 단계: 실패
2PC	로컬 트랜잭션은 아직 완료되지 않은 상태	모든 로컬 트랜잭션을 커밋	모든 로컬 트랜잭션을 취소
TC/C	모든 로컬 트랜잭션이 커밋되거나 취소된 상태로 종료	필요한 경우 새 로컬 트랜잭션 실행	이미 커밋된 트랜잭션의 실행 결과를 되돌림

TC/C 실행 도중에 지갑 서비스가 다시 시작되면 어떻게 될까?

- = 첫 번째 계정의 잔액을 갱신한 직후에 지갑 서비스가 재시작된 경우에 두 번째 계정의 잔액도 반드시 갱신되도록 하려면 어떻게 해야 할까?
- 과거 모든 작업 기록이 사라질 수 있음
- TC/C의 진행 상황(특히 각 단계 상태 정보)을 데이터베이스에 저장하는 것으로 해결 가능
    - 분산 트랜잭션의 ID와 내용
    - 각 데이터베이스에 대한 시도 단계의 상태
        - not sent yet
        - has been snet
        - response received
    - 시도 단계의 결과를 사용하여 계산할 수 있는 두 번째 단계의 이름
        - Confirm
        - Cancel
    - 두 번째 단계의 상태
    - 순서가 어긋났음을 나타내는 플래그
- 단계별 상태 테이블은 어디에 저장하면 좋을까?
    - 일반적으로 돈을 인출할 지갑의 계정이 있는 데이터베이스에 두는 것이 좋음

불균형 상태

- 시도 단계가 끝나고 나면 1달러가 사라짐
- 모든 것이 순조롭게 진행된다고 가정했을 때, 시도 단계가 끝나고 나면
    - A에서는 1달러 차감
    - C에는 변화가 없음
    - A와 C의 계정 잔액 합계 = 0달러

      ⇒ TC/C 시작 시점보다 적은 값

      ⇒ 거래 후에도 잔액 총합은 동일해야 한다는 회계 기본 원칙을 위반
- 다행히 트랜잭션 보증은 TC/C 방안에서도 유효함
- TC/C는 여러 개의 독립적인 로컬 트랜잭션으로 구성됨

    - 실행 주체 = 애플리케이션
    - 애플리케이션은 독립적 로컬 트랜잭션이 만드는 중간 결과를 볼 수 있음
- 분산 트랜잭션 실행 중에는 항상 데이터 불일치가 발생
- 데이터베이스와 같은 하위 시스템에서 불일치를 수정하는 경우에는 그 사실을 알 필요는 없지만 그렇지 않은 경우에는(=TC/C와 같은 메커니즘을 사용하는 경우에는) 직접 처리해야 함

유효한 연산 순서

- 시도 단계에서 할 수 있는 일
    - 모두 그럴듯해 보이지만 일부는 유효하지 않음

	계정 A	**계정 B**
선택 1	-$1	NOP
선택 2	NOP	+$1
선택 3	-$1	+$1

- 선택 2
    - 계정 C의 연산은 성공하였으나 계정 A에서 실패한 경우 지갑 서비스는 취소 단계를 실행해야 함
    - 취소 단계 실행 전에 누군가 C 계정에서 $1를 이미 이체하였다면, 나중에 지갑 서비스가 C에서 1달러를 차감하려고 하면 아무것도 남지 않은 것을 발견할 것

  ⇒ 분산 트랜잭션의 트랜잭션 보증을 위반
- 선택 3
    - $1를 A 계좌에서 차감하고 동시에 C에 추가하면 많은 문제가 발생할 수 있음
    - C 계좌에는 $1가 추가되었지만 A에 해당 금액을 차감하는 연산은 실패

⇒ 선택지 2, 3은 유효하지 않음

⇒ 선택지 1만이 올바른 방법

잘못된 순서로 실행된 경우

- TC/C에는 실행 순서가 어긋날 수 있다는 문제가 있음

- ex) 계정 C를 관리하는 데이터베이스에 네트워크 문제가 있어서 시도 명령 전에 취소 명령부터 받게 되는 경우
    - 그 시점에는 취소할 것이 없는 상태
    - 순서가 바뀌어 도착해도 명령을 처리할 수 있도록 하는 방법
        - 취소 명령이 먼저 도착하면 데이터베이스에 아직 상응하는 시도 명령을 못 보았음을 나타내는 플래그를 참으로 설정하여 저장해 둠
        - 시도 명령이 도착하면 항상 먼저 도착한 취소 명령이 있는지 확인 후 있다면 바로 실패를 반환

      ⇒ 단계별 상태 테이블에 순서가 어긋난 경우를 처리하기 위한 플래그를 마련했던 이유

분산 트랜잭션: 사가

- 유명한 분산 트랜잭션 솔루션 중 하나
- 모든 연산은 순서대로 정렬됨
    - 각 연산은 자기 데이터베이스에 독립 트랜잭션으로 실행됨
- 연산은 첫 번째부터 마지막까지 순서대로 실행됨
    - 한 연산이 완료되면 다음 연산이 개시됨
- 연산이 실패하면 전체 프로세스는 실패한 연산부터 맨 처음 연산까지 역순으로 보상 트랜잭션을 통해 롤백됨
    - n개의 연산을 실행하는 분산 트랜잭션을 위판 n개 연산까지 총 2n개의 연산을 준비해야 함
- 연산 실행 순서 조율
    - 분산 조율
        - 마이크로서비스 아키텍처에서 사가 분산 트랜잭션에 관련된 모든 서비스가 다른 서비스의 이벤트를 구독하여 작업을 수행하는 방식
        - 서비스가 서로 비동기식으로 통신하기 때문에 모든 서비스는 다른 서비스가 발생시킨 이벤트의 결과로 어떤 작업을 수행할지 정하기 위해 내부적으로 상태 기계를 유지해야 함
        - 서비스가 많으면 관리가 어려워짐
    - 중앙 집중형 조율

    - 하나의 조정자가 모든 서비스가 올바른 순서로 작업을 실행하도록 조율함
    - 복잡한 상황을 잘 처리하기 때문에 일반적으로 더 선호함

⇒ 어떤 방식으로 조율할지는 사업상의 필요와 목표에 따라 정함

TC/C vs 사가

	TC/C	사가
보상 트랜잭션 실행	취소 단계에서	롤백 단계에서
중앙 조정	예	예 중앙 집중형 조율 모드에서만
작업 실행 순서	임의	선형
병렬 실행 가능성	예	아니요 선형적 실행
일시적으로 일관되지 않은 상태 허용	예	예
구현 계층	애플리케이션	애플리케이션

- 실무에서는 어떤 방안이 좋을까
    - 지연 시간 요구사항이 없거나 앞서 살펴본 송금 사례처럼 서비스 수가 매우 적다 ⇒ 아무거나
        - 마이크로서비스 아키텍처에서 흔히 하는 대로 하고 싶다 ⇒ 사가
    - 지연 시간에 민감하고 많은 서비스/운영이 관계된 시스템이다 ⇒ TC/C

이벤트 소싱

배경

- 전자 지갑 서비스 제공 업체도 감사를 받을 수 있음
- 받을 수 있는 질문
    - 특정 시점의 계정 잔액을 알 수 있나요?
    - 과거 및 현재 계정 잔액이 정확한지 어떻게 알 수 있나요?
    - 코드 변경 후에도 시스템 로직이 올바른지 어떻게 검증하나요?

⇒ DDD에서 개발된 기법인 이벤트 소싱을 통해 체계적으로 답할 수 있음

정의

- 명령
  - 외부에서 전달된 의도가 명확한 요청
  - ex) 고객 A에서 C로 $1를 이체하라는 요청
  - 이벤트 소싱에서 순서는 아주 중요함 = 명령은 일반적으로 FIFO 큐에 저장됨
- 이벤트
  - 명령은 의도가 명확하지만 사실은 아니기 때문에 유효하지 않을 수도 있음
  - 유효하지 않은 명령은 실행할 수 없음
  - 작업 이행 전에는 반드시 명령의 유효성을 검사해야 함
  - 검사를 통과한 명령은 반드시 이행되어야 함
  - 이행 결과 = 이벤트
  - 명령과 이벤트의 차이점
    - 이벤트는 검증된 사실로 실행이 끝난 상태 = 이벤트에 대해서 이야기할 때는 과거 시제를 사용
    - 명령에는 무작위성이나 I/O가 포함될 수 있지만 이벤트는 결정론적 = 이벤트는 과거에 실제로 있었던 일
  - 이벤트 생성 프로세스의 특성
    - 하나의 명령으로 여러 이벤트가 만들어질 수 있음
    - 이벤트 생성 과정에는 무작위성이 개입될 수 있어서 같은 명령에 항상 동일한 이벤트들이 만들어진다는 보장이 없음
    - 이벤트 생성 과정에는 외부 I/O 또는 난수가 개입될 수 있음
    - 이벤트 순서는 명령 순서를 따라야 함 = FIFO 큐에 저장
- 상태
  - 이벤트가 적용될 때 변경되는 내용
  - 지갑 시스템에서 상태 = 모든 클라이언트 계정의 잔액
  - 맵을 자료 구조로 사용하여 표현할 수 있음
  - 키-값 저장소를 주로 사용함

- 상태 기계
  - 이벤트 소싱 프로세스를 구동함
  - 기능
    - 명령의 유효성을 검사하고 이벤트를 생성
    - 이벤트를 적용하여 상태를 갱신
  - 결정론적으로 동작해야 함
    - 무작위성을 내포할 수 없음
    - ex) I/O를 통해 외부에서 무작위적 데이터를 읽거나 난수를 사용하는 것은 허용되지 않고, 이벤트 상태에 반영하는 것 또한 항상 같은 결과로 보장해야 함

지갑 서비스 예시

- 명령
  - 이체 요청
  - FIFO 큐에 기록: 카프카
- 상태
  - 관계형 데이터베이스에 있는 계정 잔액
- 상태 기계
  - 명령을 큐에 들어간 순서대로 확인
  - 명령 하나를 읽을 때마다 계정에 충분한 잔액이 있는지 확인
  - 충분하다면 각 계정에 대한 이벤트를 만듦
  - 동작 방식
    1. 명령 대기열에서 명령을 읽음
    2. 데이터베이스에서 잔액 상태를 읽음
    3. 명령의 유효성 검사 후 유효하면 계정별로 이벤트 생성
    4. 이벤트를 읽음
    5. 데이터베이스의 잔액을 갱신하여 이벤트 적용을 마침

재현성

- 이벤트 소싱이 다른 아키텍처에 비해 갖는 가장 중요한 장점
- 이벤트를 처음부터 다시 생성하면 과거 잔액 상태는 언제든 재구성할 수 있음
- 이벤트 리스트는 불변이고 상태 기계 로직은 결정론적이기 때문에 이벤트 이력을 재생하여 만들어낸 상태는 언제나 동일함
- 재현성을 갖추면 답변할 수 있는 면접 질문
  - 특정 시점의 계정 잔액을 알 수 있나요?
    - 시작부터 계정 잔액을 알고 싶은 시점까지 이벤트를 재생하면 알 수 있음
  - 과거 및 현재 계정 잔액이 정확한지는 어떻게 알 수 있나요?
    - 이벤트 이력에서 계정 잔액을 다시 계산해보면 잔액이 정확한지 확인할 수 있음
  - 코드 변경 후에도 시스템 로직이 올바른지는 어떻게 증명할 수 있나요?
    - 새로운 코드에 동일한 이벤트 이력을 입력으로 주고 같은 결과가 나오는지 보면 됨

명령-질의 책임 분리(CQRS)

- 클라이언트는 여전히 계정 잔액을 알 수 없기 때문에 이벤트 소싱 프레임워크 외부의 클라이언트가 상태(잔액)을 알 수 있는 방법이 필요함
- 직관적인 해결법으로는 상태 이력 데이터베이스의 읽기 전용 사본을 생성한 다음 외부와 공유
- 이벤트 소싱은 상태(잔액)를 공개하는 대신 모든 이벤트를 외부에 보냄
  - 이벤트를 수신하는 외부 주체가 직접 상태를 재구축할 수 있음

  ⇒ 명령-질의 책임 분리
- 상태 기록은 담당하는 상태 기계는 하나고, 읽기 전용 상태 기계는 여러 개 일 수 있음
- 읽기 전용 상태 기계
  - 상태 뷰를 만들고 질의에 사용함
  - 이벤트 큐에서 다양한 상태 표현을 도출할 수 있음
    - 클라이언트의 잔액 질의 요청을 처리하기 위해 별도 데이터베이스에 상태를 기록
    - 이중 청구 등의 문제를 쉽게 조사할 수 있도록 하기 위해 특정한 기간 동안의 상태를 복원

- 실제 상태에 어느 정도 뒤쳐질 수 있으나 결국에는 같아짐 = 결과적 일관성 모델을 따름

상세 설계

고성능 이벤트 소싱

파일 기반의 명령 및 이벤트 목록

- 명령과 이벤트를 카프카 같은 원격 저장소가 아닌 로컬 디스크에 저장하는 방안
  - 네트워크를 통한 전송 시간을 피할 수 있음
  - 이벤트 목록은 추가 연산만 가능한 자료 구조에 저장
  - 추가는 순차적 읽기 및 쓰기 연산에서 엄청나게 최적화되어 있기 때문에 HDD에서도 잘 작동함
- 최근 명령과 이벤트를 메모리에 캐싱하는 방안
  - 메모리에 캐시해 놓으면 로컬 디스크에서 다시 로드하지 않아도 됨
- mmap 기술
  - 최적화 구현에 유용
  - 로컬 디스크에 쓰는 동시에, 최근 데이터는 메모리에 캐싱 가능
  - 디스크 파일을 메모리 배열에 대응
  - 운영체제는 파일의 특정 부분을 메모리에 캐시하여 읽기 및 쓰기 연산의 속도를 높임
  - 추가만 가능한 파일에 이루어지는 연산의 경우 필요한 모든 데이터는 거의 항상 메모리에 있기 때문에 실행 속도를 높일 수 있음

파일 기반 상태

- 상태 정보를 로컬 디스크에 저장할 수도 있음
- 파일 기반 로컬 관계형 데이터베이스 SQLite/로컬 파일 기반 키-값 저장소 Rocks DB를 사용할 수 있음
- 쓰기 작업에 최적화된 자료 구조 LSM을 사용하는 Rocks DB를 사용할 것
- 최근 데이터는 캐시하여 읽기 성능을 높임

스냅숏

- 모든 것이 파일 기반일 때 재현 프로세스의 속도를 높일 방법
  - 재현성을 확보하기 위해 상태 기계로 하여금 이벤트를 항상 처음부터 다시 읽도록 함
  - 주기적으로 상태 기계를 멈추고 현재 상태를 파일에 저장한다면 시간을 절약할 수 있음

  ⇒ 스냅숏
- 과거 특정 시점의 상태
- 변경 불가능
- 스냅숏을 저장하면 상태 기계는 더 이상 최초 이벤트에서 시작할 필요가 없음
- 스냅숏을 읽고, 어느 시점에 만들어졌는지 확인한 다음, 그 시점부터 이벤트 처리를 시작하면 됨
- 지갑 서비스 같은 금융 애플리케이션은 보통 00시 00분에 시냅숏을 찍음
  - 재무팀이 당일 발생한 모든 거래를 확인할 수 있음
- 거대한 이진 파일이며, 일반적으로는 HDFS와 같은 객체 저장소에 저장함

로컬 디스크에 데이터를 저장하는 서버는 더 이상 무상태 서버가 아닌데다, 단일 장애 지점이 된다는 문제가 있음

시스템의 안정성은 어떻게 개선할 수 있을까?

신뢰할 수 있는 고성능 이벤트 소싱

신뢰성 분석

- 데이터가 손실되면 계산 결과도 복원할 방법이 없기 때문에 데이터의 신뢰성이 중요함
- 설계하고 있는 시스템에 있는 데이터 유형
  - 파일 기반 명령
  - 파일 기반 이벤트
  - 파일 기반 상태
  - 상태 스냅숏

- 상태와 스냅숏은 이벤트 목록을 재생하면 언제든 다시 만들 수 있음
  - 상태 및 스냅숏의 안정성을 향상시키려면 이벤트 목록의 신뢰성만 보장하면 됨
- 명령어
  - 이벤트는 명령어에서 만들어지니 명령의 신뢰성만 강력하게 보장하면 충분하지 않나 생각할 수 있지만 아님
  - 이벤트 생성은 결정론적 과정이 아님
  - 난수나 외부 입출력 등의 무작위적 요소가 포함될 수 있음

  ⇒ 명령의 신뢰성 만으로는 이벤트의 재현성을 보장할 수 없음
- 이벤트
  - 이벤트는 불변이고 상태 재구성에 사용할 수 있음
  - 높은 신뢰정을 보장할 유일한 데이터

합의

- 높은 안정성을 제공하려면 이벤트 목록을 여러 노드에 복제해야 함
  - 복제 과정에서 보장해야 하는 것들
    - 데이터 손실 없음
    - 로그 파일 내 데이터의 상대적 순서는 모든 노드에 동일
- 이 목표를 달성하는 데는 합의 기반 복제 방안이 적합함
  - 모든 노드가 동일한 이벤트 목록에 합의하도록 보장함
- 래프트 알고리즘
  - 노드의 절반 이상이 온라인 상태면 그 모두에 보관된 추가 전용 리스트는 같은 데이터를 가짐
  - ex) 다섯 노드가 있을 때 래프트 알고리즘을 사용하여 데이터를 동기화하면 최소 3개 노드만 온라인 상태면 전체 시스템은 정상 동작함
  - 과반수 노드가 작동하는 한 시스템은 안정적임
  - 노드의 역할
    - 리더
      - 최대 하나의 노드만 클러스터의 리더가 되고 나머지 노드는 팔로어가 됨

        - 외부 명령을 수신하고 클러스터 노드 간에 데이터를 안정적으로 복제하는 역할을 담당
    - 후보
    - 팔로어

고신뢰성 솔루션

- 복제 메커니즘을 활용하면 파일 기반 이벤트 소싱 아키텍처에서 SPOF 문제를 없앨 수 있음
- 이벤트 소싱 노드들은 래프트 알고리즘을 사용하여 이벤트 목록을 안정적으로 동기화함
    - 리더는 외부 사용자로부터 들어오는 명령 요청을 받아 이벤트로 변환하고 로컬 이벤트 목록에 추가함
    - 래프트 알고리즘은 새로운 이벤트를 모든 팔로어에 복제함
    - 팔로어를 포함한 모든 노드가 이벤트 목록을 처리하고 상태를 업데이트함
    - 래프트 알고리즘은 리더와 팔로어가 동일한 이벤트 목록을 갖도록 하며, 이벤트 소싱은 동일한 이벤트 목록에서 항상 동일한 상태가 만들어지도록 함
- 안정적인 시스템은 장애를 원활하게 처리해야 함
    - 리더에 장애 발생
        - 래프트 알고리즘은 나머지 정상 노드 중에서 새 리더를 선출
        - 새 리더는 외부 사용자로부터 오는 명령을 수신할 책임을 짐
        - 한 노드가 다운되어도 클러스터는 계속 서비스를 제공할 수 있음
        - 리더 장애가 명령 목록이 이벤트로 변환되기 전에 발생할 수 있음
            - 클라이언트는 시간 초과/오류 응답을 받음
            - 클라이언트는 새로 선출된 리더에게 같은 명령을 다시 보내야 함
    - 팔로어에 장애 발생
        - 해당 팔로어로 전송된 요청은 실패함
        - 래프트는 죽은 노드가 다시 시작되거나 새로운 노드로 대체될 때까지 기한 없이 재시도하여 해당 장애를 처리함

분산 이벤트 소싱

- 안정적인 고성능 이벤트 소싱 아키텍처를 구현하는 방법은 신뢰성 문제는 해결하지만 다른 문제가 있음
    - 전자 지갑 업데이트 결과는 즉시 받고 싶은데 클라이언트가 디지털 지갑의 업데이트 시점을 정확히 알 수 없어서 주기적 폴링에 의존해야 할 수 있기 때문에 CQRS 시스템에서는 요청/응답 흐름이 느릴 수 있음
    - 단일 래프트 그룹의 용량이 제한되어 있기 때문에 일정 규모 이상에서는 데이터를 샤딩하고 분산 트랜잭션을 구현해야 함

풀 vs 푸시

- 풀 모델
    - 외부 사용자가 읽기 전용 상태 기계에서 주기적으로 실행 상태를 읽음
    - 실시간이 아님
    - 읽기 주기를 너무 짧게 설정하면 지갑 서비스에 과부하가 걸릴 수 있음
    - 외부 사용자와 이벤트 소싱 노드 사이에 역방향 프락시를 추가하면 개션 가능
        - 외부 사용자는 역방향 프락시에 명령을 보냄
        - 역방향 프락시 명령을 이벤트 소싱 노드로 전달하는 한편 주기적으로 실행 상태를 질의함
        - 여전히 통신이 실시간으로 이루어지지는 않지만 다만 클라이언트의 로직은 단순해짐
- 푸시 모델
    - 역방향 프락시를 두고 나면 읽기 전용 상태 기계를 수정하여 응답 속도를 높일 수 있음
    - 읽기 전용 상태 기계가 자기만의 특별한 로직을 가질 수 있음
    - 읽기 전용 상태 기계로 하여금 이벤트를 수신하자마자 실행 상태를 역방향 프락시에 푸시하도록 하여 사용자에게 실시간으로 응답이 이루어지는 느낌을 줄 수 있음

분산 트랜잭션

- 모든 이벤트 소싱 노드 그룹이 동기적 실행 모델을 채택하면 TC/C나 사가 같은 분산 트랜잭션 솔루션을 재사용할 수 있음

- 본 설계안에서는 키의 해시 값을 2로 나누어 데이터가 위치할 파티션을 정한다고 가정함

최종 분산 이벤트 소싱 아키텍처에서 이체가 이루어지는 방법

1. 사용자 A가 사가 조정자에게 분산 트랜잭션을 보냄
    - A: -$1
    - C: +$1
2. 사가 조정자는 단계별 상태 테이블에 레코드를 생성하여 트랜잭션 상태를 추적함
3. 사가 조정자는 작업 순서를 검토한 후 [A: -$1]을 먼저 처리하기로 결정함
    - 조정자는 [A: -$1]의 명령을 계정 A 정보가 들어 있는 파티션 1로 보냄
4. 파티션 1의 래프트 리더는 [A: -$1] 명령을 수신하고 명령 목록에 저장함
    - 명령의 유효성을 검사함
    - 유효하면 이벤트로 변환됨
    - 래프트 합의 알고리즘은 여러 노드 사이에 데이터를 동기화하기 위한 것
    - 동기화가 완료되면 이벤트가 실행됨
5. 이벤트가 동기화되면 파티션 1의 이벤트 소싱 프레임워크가 CQRS를 사용하여 데이터를 읽기 경로에 동기화함
    - 읽기 경로는 상태 및 실행 상태를 재구성함
6. 파티션 1의 읽기 경로는 이벤트 소싱 프레임워크를 호출한 사가 조정자에 상태를 푸시함
7. 사가 조정자는 파티션 1에서 성공 상태를 수신함
8. 사가 조정자는 단계별 상태 테이블에 파티션 1의 작업이 성공했음을 나타내는 레코드를 생성함
9. 첫 번째 작업이 성공했으므로 사가 조정자는 두 번째 작업인 [C: +1$]을 실행함
    a. 조정자는 계정 C의 정보가 포함된 파티션 2에 [C: +$1] 명령을 보냄
10. 파티션 2의 래프트 리더가 [C: +$1] 명령을 수신하여 명령 목록에 저장함
    - 유효한 명령으로 확인되면 이벤트로 변환됨
    - 래프트 합의 알고리즘이 여러 노드에 데이터를 동기화 함
    - 동기화가 끝나면 해당 이벤트가 실행됨

11. 이벤트가 동기화되면 파티션 2의 이벤트 소싱 프레임워크는 CQRS를 사용하여 데이터를 읽기 경로로 동기화함
    a. 읽기 경로는 상태 및 실행 상태를 재구성함
12. 파티션 2의 읽기 경로는 이벤트 소싱 프레임워크를 호출한 사가 조정자에 상태를 푸시함
13. 사가 조정자는 파티션 2로부터 성공 상태를 받음
14. 사가 조정자는 단계별 상태 테이블에 파티션 2의 작업이 성공했음을 나타내는 레코드를 생성함
15. 이때 모든 작업이 성공하고 분산 트랜잭션이 완료됨
    - 사가 조정자는 호출자에게 결과를 응답함

마무리

참 잘했어요~! 👏