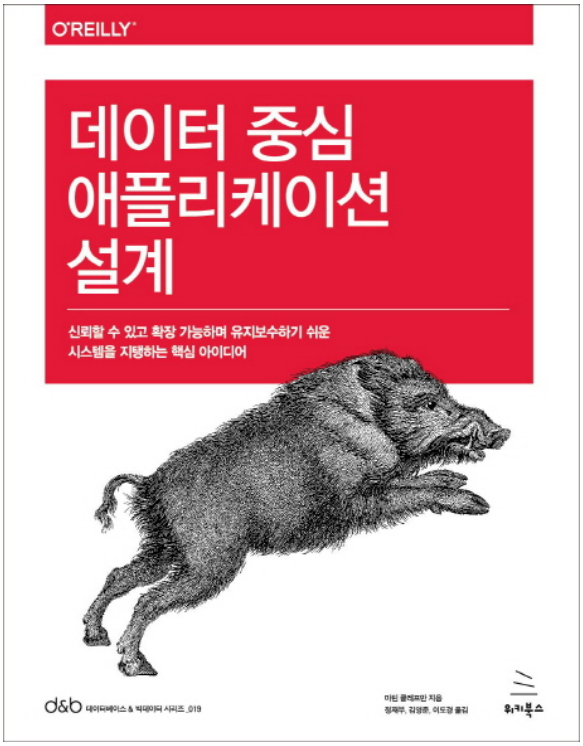
데이터 중심 애플리케이션 설계 - 9장. 일관성과 합의



귀여운 멧돼지

9장. 일관성과 협의

여는 글

결함을 다루는 가장 간단한 방법은 그냥 전체 서비스가 실패하도록 두고 사용자에게 오류 메시지를 보여주는 것이다.

만약 싫다면..? 견뎌낼 방법(tolerating)을 찾아야 하며, 내부 구성 요소 중 뭔가에 결함이 있더라도 서비스는 올바르게 동작하게 할 방법을 찾아야 한다. 이를 **내결함성**이라고 한다.

운영을 하다보면 노드가 멈추거나 네트워크 패킷이 손실되거나 순서가 뒤섞이거나, 중복되거나, 지연이 발생하거나 여러 상황을 마주 할 수 있다.

내결함성을 보장해주는 가장 좋은 방법은 범용 추상화를 찾아 이를 구현하고 애플리케이션에서 이 보장에 의존하게 하는 것이다.

→ 내결함성을 보장하기 위한 방법을 찾아 적용하고 애플리케이션이 이에 의존하도록 하자!

→ 추상화 : 멀티 클러스터로 구성되어 있더라도 하나만 있는것 처럼!

내결함성을 위한 추상화(방법)중 대표적인 것이 **합의**이다. 합의는 모든 노드가 어떤 것에 동의하게 만드는 것이다. 장애 발생시에도 합의에 도달하는 것은 어려운 일이다. **단일 리더 복제** 상황에서 리더가 죽으면 **남는 노드**들간에 **합의**를 통해 모든 노드가 동의하는 새 리더를 선출함으로써 이어갈 수 있다. 근데 두 개 이상의 노드가 리더인줄 착각하는 상황을 **스플릿 브레인(split brain)**이라고 하며 이는 곧 **데이터 손실**로 이어진다.

중요한 것은 어떤 것을 할 수 있고 할 수 없는지에 대한 범위를 이해해야 한다. 어떤 상황에서 시스템이 결함을 견뎌내고 계속 동작할 수 있지만 어떤 상황에서는 불가능하다.

일관성 보장

불일치 문제는 리더 없음, 단일 리더, 다중 리더 모든 상황에서 발생할 수 있다. (163p)

복제 데이터베이스는 대부분 최소한 **최종적 일관성**을 제공한다. 데이터베이스에 쓰기를 멈추고 불특정시간 동안 기다리면 결국 모든 읽기 요청이 같은 값을 반환한다는 뜻이다. 바꿔 말하면 불일치는 **일시적** 이며 결국 스스로 해소한다. 이런 점에서 **최종적 일관성**보단 **수렴**이란 표현이 적절할 수도 있겠다.

그러나 막연히 언제가라고 할 뿐 언제 수렴될지에 대해서는 보장해주지 않는다.

약한 보장만 제공하는 데이터베이스를 다룰 때는 그 제한을 계속 알아야 하고 뜻하지 않게 너무 많은 것을 가정하면 안된다.

→ 신뢰도와 역량(?)에 맞는 책임 부여의 중요성!

성능 ↔ 내결함성, 성능, 보장

이 장에서는 여러 일관성 모델에 대해 설명하는데 이것은 공짜가 아니며 트레이드 오프가 있고, 성능과 일관성 그리고 내결함성은 **반비례 관계**를 가진다. 강한 보장을 하는 모델은 성능

이 나쁘고, 약한 보장을 하는 서비스는 내결함성이 약할 수 있다.

여러 모델을 알아두고 적절히 선택 하는 것이 중요하다.

지금부터 가장 강한 일관성 모델 중 하나인 **선형성(linearizability**)가 무엇이고 장, 단점이 무엇인지, **이벤트 순서화 문제**와 **인과성**, **순서화 문제**는 무엇인지, **분산 트랜잭션을 원자적으로 커밋**하는 방법이 무엇인지에 대해 차례대로 알아보자.

선형성

최종적 일관성을 지닌 DB에서 두 개 의 다른 복제본에 같은 질문을 동시에 하면 두 가지 다른 응답을 받을 지도 모르며, 많은 혼란을 느낄 지도 모른다.

그래서 좀 더 명쾌하게 하기 위해 데이터베이스가 하나의 복제본이 있으면 좋을 것이다.

결론적으로, **선형성**은 시스템에 데이터 복사본이 하나만 있고 그 데이터를 대상으로 수행하는 모든 연산은 원자적인 것처럼 보이게 만드는 것이다. 그래서 선형성이 있으면 수정된 최신 데이터는 즉시 조회가 가능하기 때문에 **최신성 보장(recency guarantee)**이란 특성이 있다.

책에서는 선형성이 없는 것에 대한 예시로 리더와 두 개의 팔로워를 운영하는 스포츠 웹사이트에 경기 두 사용자가 결과 조회를 요청한 상황을 들었다. 이 때 각 사용자는 서로 다른 팔로워에게 응답을 받고, 이 때 결과를 받았다. 선형성이 있다면 동일한 결과를 받았을 것이다.

선형성과 직렬성

약간은 비슷하게 보일 수 있는 두 개념의 정의는 다음과 같다.

> 직렬성: 모든 트랜잭션들이 여러 객체를 읽고 쓸 수 있는 상황에서의 트랜잭션들의 격리 속성이다. 직렬성은 트랜잭션의 실행 순서와 상관없이 트랜잭션이 어떤 순서에 따라 실행하는 것처럼 동작하게 보장해준다.

> 선형성: 레지스터 (개별 객체)에 실행되는 읽기와 쓰기에 대한 보장이다. 연산을 트랜잭션으로 묶지 않아서 충돌 구체화 같은 수단을 쓰지 않으면 쓰기 스큐같은 문제를 막지 못한다. (읽은 값 기반으로 쓰지만, 쓰는 시점에는 그 값이 참이 아님)

데이터베이스는 두 가지를 모두 제공할 수 있고 그 조합은 엄격한 직렬성이나 강한 단일 복사본 직렬성이라고 한다. 2PL(2 Phase Lock)은 선형적이지만, 7장에 나오는 SSI는 선형적이지 않다.

선형성에 기대기

선형성이 유용한 상황이 몇가지 있다.

잠금과 리더 선출

- 단일 리더 복제를 사용하는 시스템은 리더가 하나만 있도록 보장해야한다. 한 방법은 잠금을 사용하는 것인데, 이 잠금을 구현할 때는 선형적이어야한다. 모든 노드는 어느 노드가 잠금을 소유할지 동의해야한다.
- 분산 잠금과 리더 선출에는 아파치 주키퍼나 etcd와 같은 코디네이션 서비스가 종종 사용된다. 이 서비스에서도 선형성 저장소 서비스가 기초가된다.

제약조건과 유일성 보장

- unique constraint 는 데이터베이스에서 흔하다. 이 상황은 실제로 잠금과 비슷하고, 연산 자체가 compare-and-set과도 비슷하다. 은행 계좌 잔고, 영화좌석과 같은 경우도 제약이 있다. 이 제약들은 **모든 노드가 동의하는 하나의 최신값**을 요구한다.
- 이런 경우 선형성이 필요하다. 외래키나 속성 제약같은 건 선형성 없이도 구현할 수 있다.

채널 간 타이밍 의존성

- 썸네일 시스템을 생각해보자. 사용자가 이미지를 올리면, 다른 사용자가 쉽게 받을 수 있도록 저해상도로 만드는 시스템이다. 이 썸네일링 명령은 메시지 큐를 통해 웹서버에서 크기 변경 모듈로 보내진다. 이 시스템이 선형적이지 않으면, 메시지 큐의 단계가 저

장서비스 내부의 복제보다 빠를 수 있다. 그러면 메시지를 받은 모듈이 이미지를 불러올 때 불러오지 못하거나 이전 데이터만 볼 수 있다.

- 이 문제는 웹서버와 이미지 크기 변경 모듈 사이에 두가지 채널, 즉 파일 저장소와 메시지 큐가 있기때문에 발생한다. 선형성의 최신 보장이 없으면 경쟁 조건이 발생할 수 있다.

선형적 시스템 구현하기

다시 한 번 **"선형성은 데이터 복사본이 하나만 있는 것처럼 동작하고 그 데이터에 실행하는 모든 연산은 원자적이라는 것"**을 의미힌다.

그런데 복사본이 하나이면 결함을 견뎌낼 수가 없다. 시스템이 내결함성을 지니도록 만드는 가장 흔하고 확실한 방법은 복제를 사용하는 것이다.

단일 리더 복제(선형적이 될 가능성이 있음)

단일 리더 복제를 하는 시스템에서는 리더는 쓰기에 사용되는 데이터의 주 복사본을 갖고 있고 팔로워는 다른 노드에 데이터의 백업 복사본을 보관한다. 리더나 동기식으로 갱싱된 팔로워에서 실행한 읽기는 선형적이 될 가능성이 있다. 그러나 모든 단일 리더 데이터베이스가 실제로 선형적인 것은 아니다. 설계 때문일 수도 있고 동시성 버그 때문일 수도 있다.

합의 알고리즘(선형적)

합의 알고리즘은 단일 리더 복제를 닮았지만, 합의 프로토콜에는 스플릿 브레인과 복제본이 뒤처지는 문제를 막을 수단이 포함된다. 이런 세부 사항 덕에 합의 알고리즘은 선형성 저장소를 안전하게 구현할 수 있다. 대표적으로 etcd, 주키퍼가 있다.

다중 리더 복제(비선형적)

여러 노드에서 동시에 쓰기를 처리하고 그렇게 쓰여진 내용을 비동기로 다른 노드에 복제한다. 이런 까닭으로 다중 리더 복제 시스템은 충돌 해소가 필요한 충돌 쓰기를 만들 수 있다.

리더 없는 복제(아마도 비선형적)

정족수를 통해서 엄격한 일관성을 달성할 수 있다고 주장하는 사람들이 있다. 그러나 정족수 설정과 일관성의 정의에 따르면 이 주장이 완벽한 진실은 아니다.

선형성과 정족수

다이나모 스타일(리더 없는 복제)에서 엄격한 정족수를 사용한 읽기, 쓰기는 선형적인 것처럼 보인다. 그러나 네트워크 지연이 심하면 경쟁조건이 생길 수 있다.

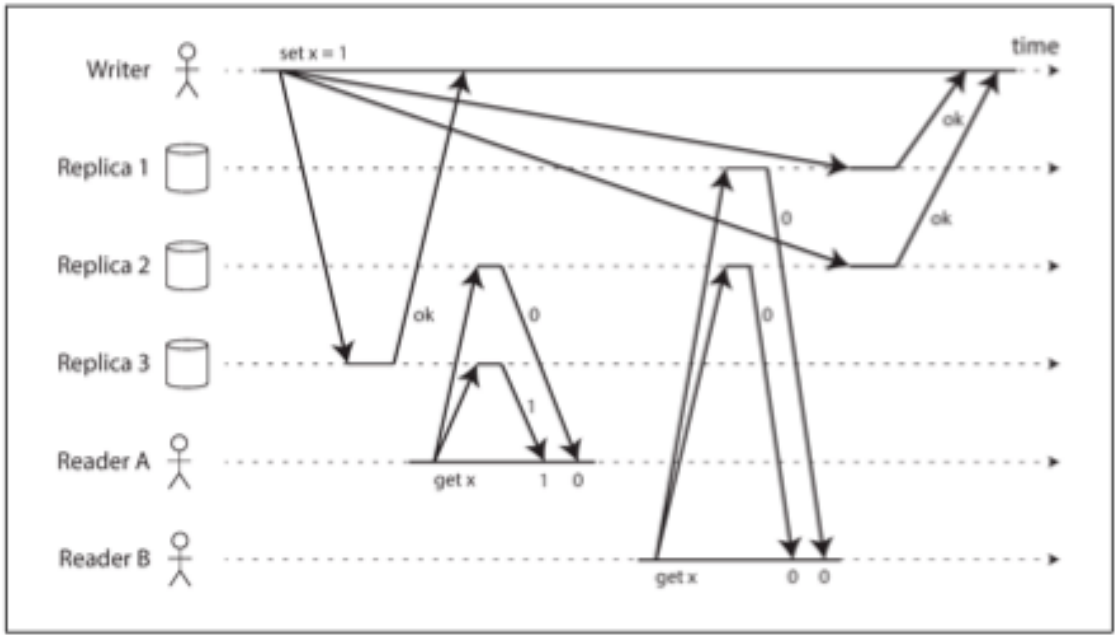


Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

정족수가 만족 (w=3, r=2, n=3)되지만 A가 1을 보고, B는 0을 본다. 이런 경우 선형성을 만족하지 않는다. 그러나 성능을 비용으로 지불하고 선형적으로 만드는게 가능하다.

- 동기식 읽기 복구
- 쓰기는 쓰기 전 최신 상태 읽기

이 방법으로는 선형성 읽기와 쓰기 연산만 구현할 수 있고 선형성 compare-and-set 연산은 **합의 알고리즘**이 필요하므로 구현할 수 없다. 그러므로 다이나모 스타일은 **선형성**을 만족하지 않는다고 보는게 안전하다.

선형성의 비용

다중 리더 데이터베이스를 생각해보자. 만약 데이터 센터 간 연결할 수 없게 되면, 각 데이터 센터는 따로 정상 동작한다. 서로 동기화 안된 부분은 네트워크가 복구되면 따라 잡을 수 있다.

그러나 단일 리더에서 데이터 센터가 분리된 예제에서는, 팔로워 데이터 센터로 접속한 클라이언트가 쓰기, 선형성 읽기와 를 할 수 없다.

모든 선형성 데이터베이스는 이런 문제가 있다. 이 트레이드 오프를 CAP 정리로 표현한다.

CAP 정리

- 애플리케이션에서 **선형성**을 요구하고, 네트워크 문제때문에 복제서버가 다른 복제서버와 연결이 끊기면 일부 복제 서버는 연결이 끊긴 동안은 **요청을 처리할 수 없다. (CP)**
- 애플리케이션에서 선형성을 요구하지 않는다면, 독립적인 방식으로 요청을 처리할 수 있고 따라서 **가용**한 상태를 유지한다. **(AP)**

> CAP 은 이 트레이드 오프를 이야기한다. Consistency(일관성), Availability(가용성), Partition tolerance(분단 내성)이라는 셋 중 둘을 고르라는 식의 해석이 있지만 오해의 소지가 있다. 이는 사실 네트워크 분단이 생겼을 때 일관성과 가용성 중 하나를 고르라는 의미로 보는 것이 좋다.

선형성과 네트워크 지연

선형성은 유용한 보장이지만 현실에서 실제로 선형적인 시스템은 매우 드물다. 예를 들어, 최신 다중 코어의 RAM 조차 선형적이지 않다. 하나의 CPU 코어에서 실행 중인 스레드가 메모리 주소에 쓴 후 바로 다른 코어에서 실행되는 스레드가 같은 주소를 읽으면 처음에 읽은 값을 읽으라는 보장이 없다.

이는 모든 CPU 코어가 저마다의 메모리 캐시와 저장 버퍼를 사용하기 때문이다. 캐시가 물론 저장소보다 빠르므로 좋은 성능에 필수적이고, 각자의 저장소를 바라보는 결과를 낳는다. 여기에서 선형성을 제거한 이유는 내결함성이 아니라 **성능**이다.

선형성은 느리다. 그리고 이는 네트워크 결함이 있을 때뿐만 아니라 항상 참이다. 좀더 효율적인 선형 저장소를 찾으려는 시도가 있었지만, 아직까지는 답이 없다. 다만 완화된 일관성 모델은 훨씬 더 빠를 수 있다. 따라서 지연 시간에 민감한 시스템에서는 이 트레이드 오프가 중요하다.

순서화 보장

앞에서 **선형성 레지스터**는 **데이터 복사본이 하나만 있는 것 처럼 동작하고 모든 연산이 어느 시점에 원자적으로 효과가 나타나는 것처럼 보인다고 했다.** 이 정의는 연산들이 어떤 잘 정

의된 순서대로 실행되는 것을 암시한다. 순서는 이 책에서 되풀이된 주제이며 이는 순서화가 중요한 근본적 아이디어일 수도 있다는 것을 시사한다.

- 5장 단일 리더 복제에서 리더의 주 목적은 복제 로그의 쓰기 순서를 결정하는 것이라고 배웠다. 단일 리더가 없으면 동시에 실행되는 연산 때문에 충돌이 발생한다.
- 7장 직렬성에서 트랜잭션이 어떤 일련 순서에 따라 실행되는 보장하는 것과 관련돼 있다.
- 8장에서의 타임스탬프와 시계는 질서를 부여하려는 시도다.

순서화와 인과성

- 일관된 순서로 읽기 : 질문과 답변이라는 두 개의 처리는 질문이 선이고 답변이 후이다. 이처럼 두 처리는 **인과적 의존성(causal dependency)**이 있다고 한다.
- 네트워크 지연때문에 존재 해야 할 로우가 없어 존재하지 않는 로우를 갱신하는 예제.
- 동시 쓰기 감지에서 **이전 발생(happened before)** 관계를 얘기했다. A < B (시간적으로) 라면, B가 A에 의존하거나 기반으로 한다는 뜻이다. 동시적이면 서로에 대해서 모른다는 뜻이다.
- 트랜잭션 스냅숏 격리의 맥락에서 일관적 스냅숏을 이야기했다. 일관적은 인과성에 일관적이라는 의미다. 스냅숏에 답변이 포함되면 응답된 질문 또한 포함되어야한다.
- 쓰기 스큐 예제에서도 인과적 의존성을 보여준다. SSI 는 트랜잭션 사이의 인과적 의존성을 추론해서 쓰기 스큐를 검출한다. (온 콜 의사 예제)
- 이 장에 초반에 나오는 스포츠 웹사이트 예제 또한 특정 사용자가 결과를 확인해야 다른 사용자가 확인할 수 있다는 것은 의존성이 있다. (채널 간 타이밍 의존성)

시스템이 인과성에 의해 부과된 순서를 지키면 그 시스템은 인과적으로 일관적(causally consistent)이라고 한다.

인과적 순서가 전체 순서는 아니다

전체 순서는 어떤 두 요소를 비교할 수 있게하므로 두 요소가 있으면 항상 어느 것이 크고 작은지 비교할 수 있다. 이제 집합을 생각해보자. 집합의 크기는 비교가 불가능하다. {a, b} 와 {b,c} 중 어느것이 크고 작은가? 집합은 부분적으로 순서가 정해진다(partially ordered) 이것이 부분 순서이다.

이 차이는 데이터베이스 일관성 모델에도 반영된다.

- 선형성

    - 선형성 시스템에서는 **연산의 전체 순서를 정할 수 있다**. DB 복제본이 하나만 있는 것 처럼 동작하고 연산이 원자적이면 어떤 두 연산에 대해서 항상 둘 중 하나가 먼저 실행되었다고 말할 수 있다.
- 인과성
    - 두 연산 중 어떤 것도 다른 것보다 먼저 실행되지 않았다면 **동시적**이라고 말한다. 두 연산이 동시에 실행되면 비교할 수 없다. 인과성은 부분 순서를 정의한다.

이 정의에 따르면 **선형성은 동시적 연산이 없다**. 하나의 타임라인이 있고, 모든 연산은 해당 타임라인을 따라 순서가 정해져야한다.

동시성은 타임라인이 갈라졌다 다시 합쳐지는 것을 말한다. 이 경우 다른 branch 에 있는 연산은 비교가 불가능하다.

선형성은 인과적 일관성보다 강하다

그러면 이런 결론에 다다른다. 선형성은 인과성을 내포한다. 어떤 시스템이든지 선형적이라면 일관성도 올바르게 유지한다. 좋은 소식처럼 보이지만, 앞서 말했듯이 **선형성은 비용이 비싸다.**

좋은 소식은 절충이 가능하다는 것이다. 선형성은 인과성을 유지하는 유일한 방법은 아니고 성능 손해를 유발하지 않고도 인과적 일관성을 유지할 수 있다. 이 연구는 현재 계속 진행되고 있다.

인과적 의존성 담기

이 인과적 의존성의 핵심 아이디어는 다음과 같다,

- 일련번호 순서화: 타임스탬프나 논리적 시계, 모든 연산마다 증가하는 카운터 등을 통해 순서를 정한다.
- 비인과적 일련번호 생성기
    - 노드 별 일련번호 집합 사용
    - wallclock time
    - 일련번호 블록 사전 할당
- 렘포트 타임스탬프: 카운터, 노드ID를 활용하는 방법

타임스탬프 순서화로는 충분하지 않다

램포트 타임스탬프가 전체 순서를 정의해주긴하지만, 분산시스템의 공통 문제를 해결하기엔 부족하다.

예를 들어 유니크한 사용자 명을 갖도록 보장하는 시스템을 고려해보자. 두 사용자가 동시에 같은 이름으로 만들려고 하면 하나는 성공하고 하나는 실패한다. 사용자가 둘다 생성된 이후에 타임스탬프로 하나를 실패하게 할 수는 있다. (사후 처리) 그러나 노드가 사용자로부터 유저네임 생성 요청을 받고 그 요청을 당장 성공/실패 처리해야하는 경우라면 이 노드는 다른 노드가 같은 유저네임 생성을 하고 있는지, 어떤 타임스탬프를 배정받았는지 알지 못한다.

이를 알기 위해서는 **다른 노드가 뭘 하고 있는지 확인**해야한다. 이를 구현하면 **한 노드만 장애가 생겨도 시스템이 멈추게 된다.** 이는 **내결함성**에 좋지 않다.

즉 문제는 전체 순서가 모든 연산을 모은 이후에 드러난다는 것이다. 그러므로 유일성 제약 조건 같은 것을 구현하려면 전체 순서로 충분하지 않다. 언제 전체순서가 확정되는지도 중요하다.

그래서 언제 전체 순서가 확정되는지 알아야 한다는 아이디어를 **전체 순서 브로캐스트의** 주제로 다루겠다.

전체 순서 브로드캐스트

위에서 언제 전체 순서가 확정되는지에 대한 아이디어를 이야기했다. 프로그램이 단일 CPU 코어에서 실행된다면 연산의 전체 순서를 정하기 쉽다. 하지만 분산 시스템에서는 위와 같이 타임스탬프나 일련번호를 이용할 수 있지만, 문제가 있다는 것을 발견했다.

단일 리더 상황에서 리더가 처리할 수 있는 수준을 넘어설 수 있는데, 이 때 시스템을 확장하고 리더에 발생한 장애를 어떻게 복구할 것인가의 문제를 **전체 순서 브로드캐스트(혹은 원자적 브로드캐스트)** 라고 한다. 이는 노드 사이에 메시지를 교환하는 프로토콜로 기술된다.

- 신뢰성 있는 전달
  - 어떤 메시지도 손실되지 않는다. 한 노드에만 메시지가 전달되면 모든 노드에도 전달
- 전체 순서가 정해진 전달
  - 메시지는 모든 노드가 같은 순서로 전달된다.

전체 순서 브로드캐스트를 구현하는 올바른 알고리즘은 위 **두 조건**을 항상 **만족**해야한다.

전체 순서 브로드캐스트 사용하기

주키퍼나 etcd와 같은 합의 서비스는 전체 순서 브로드캐스트를 실제로 구현한다.

전체 순서 브로드 캐스트는 복제에 필요한 것이다. 모든 메시지가 DB에 쓰기를 나타내고, 모든 복제 연산이 같은 순서로 처리하면 복제 서버들은 서로 일관성 있는 상태를 유지한다. 이 원리를 상태 기계 복제라고한다. (state machine replication)

마찬가지로 전체 순서 브로드캐스트를 **직렬성 트랜잭션**을 구현하는데도 쓸 수 있다. 모든 메시지가 스토어드 프로시저로 실행되는 결정적 트랜잭션을 나타낸다면, 그리고 모든 노드가 이를 순서대로 처리한다면 데이터베이스의 파티션과 복제본은 서로 일관적인 상태를 유지한다.

중요한 측면은 전체 순서 브로드캐스트는 **메시지가 전달되는 시점**에 그 **순서가 고정**된다는 것이다. 나중에 도착된 메시지가 **중간에 끼어들거나** 소급적용되는것이 없다. 이때문에 앞서 말한 타임스탬프 순서화보다 강하다.

전체 순서 브로드캐스트는 **펜싱 토큰**을 제공하는 **잠금 서비스**를 구현하는데도 유용하다. 잠금을 획득하는 모든 요청은 메시지로 로그에 추가되고, 모든 메시지는 로그에 나타난 순서대로 일련번호가 붙는다. 일련번호는 단조 증가하므로 펜싱 토큰의 역할을 할 수 있다. 주키퍼에서는 이를 zxid라고 한다.

전체 순서 브로드캐스트를 사용해 선형성 저장소 구현하기

전체 순서 브로드캐스트는 **비동기식**이다. 메시지는 **고정된 순서**로 **신뢰성** 있게 전달되지만 언제 전달될지는 보장되지않는다. 그러나 **선형성**은 **최신성 보장**이다. **읽기가 최근에 쓰여진 값을 보는게 보장**된다.

그러나 전체 순서 브로드캐스트 구현이 있다면 이를 기반으로 한 **선형성 저장소**를 만들 수 있다. 예를 들어 unique constraint 이 있다.

모든 사용자 명마다 원자적 compare and set 연산이 구현된 선형성 저장소를 가질 수 있다고 상상해보자. 모든 레지스터는 초기에 null이다 . 사용자명을 생성할 때 null 인 경우만 set 할 수 있도록 한다. 선형성때문에 여러 유저가 같은 사용자명을 가지려고하면 compare-and-set 연산 중 하나만 성공한다.

전체 순서 브로드캐스트를 추가 전용 로그로 사용해 선형성 compare-and-set 연산을 다음과 같이 구현할 수 있다.

- 메시지를 로그에 추가해서 점유하기 원하는 사용자명을 시험적으로 가리킨다
- 로그를 읽고, 추가한 메시지가 되돌아오기를 기다린다.
- 원하는 사용자명을 점유하려고하는 메시지가 있는지 확인한다. 본인이 첫 메시지라면 성공이다. 다른 사용자가 보낸 메시지라면 연산을 어보트 시킨다.

로그 항목은 모든 노드에 같은 순서로 전달되므로 여러 쓰기가 동시에 실행되면 모든 노드가 어떤 쓰기가 먼저 실행된 것인지 동의한다.

이 절차는 선형성 쓰기를 보장하지만, 선형성 읽기는 보장하지않는다. 로그로부터 비동기로 갱신되는 저장소를 읽으면 오래된 값이 읽힐 수 있다. 읽기를 선형적으로 만들려면 몇가지 선택지가 있다.

- 로그를 통해 순차 읽기를 할 수 있다. 로그에 메시지를 추가하고 로그를 읽어서 메시지가 되돌아왔을 때 실제 읽기를 수행하면 된다. 따라서 로그상의 메시지 위치는 읽기가 실행된 시점을 나타낸다.
- 로그에서 최신 로그 메시지의 위치를 선형적 방법으로 얻을 수 있다면 그 위치를 질의하고 그 위치까지의 모든 항목이 전달되기를 기다린 후 읽기를 수행할 수 있다.
- 쓰기를 실행할 때 동기식으로 갱신돼서 최신이 보장되는 복제 서버에서 읽을 수 있다.

선형성 저장소를 이용해 전체 순서 브로드캐스트 구현하기

선형성 저장소가 있을 때 전체 순서 브로캐스트를 구현하는 것도 가능하다. 가장 쉬운 방법은 정수를 저장하고 원자적 increment-and-get연산이 지원되는 선형성 레지스터가 있다고 가정하는 것이다.

알고리즘은 간단하다. 전체 순서 브로드캐스트를 통해 보내고 싶은 모든 메시지에 대해 선형성 정수로 increment-and-get 연산을 수행하고 레지스터에서 얻은 값을 일련번호로 메시지에 붙인다. 그 후 메시지를 모든 노드에 보낼 수 있고 수신자들은 일련번호 순서대로 메시지를 전달한다.

실패가 없다면 이 방법은 제대로 동작한다. 메시지 4를 전달하고, 일련번호가 6인 메시지를 받았다면 5를 기다려야한다는 것을 알 수 있다. 문제는 노드에 장애나 네트워크 오류가 발생했을 때 일어난다. 이렇게 일련번호 생성에 대해서 고민하다보면 필연적으로 합의 알고리즘에 도달하게 된다. 이는 우연이 아니다.

분산 트랜잭션과 합의

합의는 분산 컴퓨팅에서 가장 중요하고 근본적인 문제 중 하나다. 목적은 노드들이 뭔가에 동의하게 만드는 것이다. 이 문제는 생각만큼 간단하진 않다.

노드가 동의하는 것이 중요한 상황이 여럿 있다.

- 리더 선출
    - 단일 리더 복제를 사용하는 DB에서 모든 노드는 어떤 노드가 리더인지 동의해야한다. 어떤 노드가 네트워크 결함때문에 통신이 불가능하면, 리더십 지위를 놓고 경쟁

할 수 있다.

- 원자적 커밋
  - 여러 노드나 파티션에 걸친 트랜잭션을 지원하는 DB에서는... 트랜잭션이 어떤 노드에서는 성공하고, 어디서는 실패하는 문제가 있을 수 있다. 원자성을 유지하고 싶다면 노드들이 성공/실패 여부에 합의하게 만들어야한다.

> **합의불가능성** 피셔, 린치, 패터슨은 FLP 라는 결과를 내놓았다. 이 내용인 즉슨 어떤 노드가 죽을 위험이 있다면 항상 합의에 다다를 알고리즘은 없다는 것이다. 아니 그럼 왜 이걸 배워요. 사실 이 결과는 어떤 시계나 타임아웃도 사용할 수 없는 경우에 증명/적용되는 모델이다. 타임아웃이 있거나 죽은 노드를 살펴볼수있는 방법이 있다면 보통의 현실에서 분산시스템은 합의를 달성할 수 있다.

원자적 커밋과 2단계 커밋

트랜잭션 원자성의 목적은 여러 쓰기 중 잘못되는 경우 간단한 시맨틱을 제공하기 위함이다. 트랜잭션의 결과는 커밋 성공이나 어보트다.

이 원자성은 특히 다중 객체 트랜잭션과 보조 색인을 유지하는 데이터베이스에서 중요하다. 개별 보조색인은 주 데이터와 분리된 자료이고, 주 데이터가 변경되면 그 내용이 반영되어야 한다.

단일 노드에서 분산 원자적 커밋으로

단일 노드에서는 커밋 요청을 받으면 디스크 로그에 커밋 레코드를 추가한다. 이 과정에서 DB가 죽으면 트랜잭션은 노드가 재시작될때 로그로 복구하고, 죽기전에 레코드가 쓰였다면 커밋이 된것이다. 그러니까, 단일 노드에서 커밋은 데이터가 디스크에 지속성 있게 쓰여지는 순서에 의존한다. 데이터가 먼저고 커밋 레코드는 그 다음이다. (어보트 될 가능성이 있지만, 이 시점에서는 커밋이다.)

그런데 트랜잭션에 여러 노드가 관여한다면 어떨까? 예를 들어, 파티셔닝된 데이터베이스에서 다중 객체 트랜잭션을 쓰거나, 보조 색인을 사용할수도 있다.

이런 경우 모든 노드에 커밋 요청을 보내고 각 노드에서 트랜잭션을 독립적으로 지원하는 것으로는 충분치 않다. 언제 누가 죽을지 몰라... 이러면 한 노드는 성공, 다른 노드는 실패하면서 원자성을 위반하기 쉽다.

트랜잭션 커밋은 되돌릴 수 없어야한다. 데이터가 커밋되면 다른 트랜잭션에게 보이게 되고 다른 클라이언트도 이 데이터에 의존하기 시작할지도 모르기때문이다.

2단계 커밋 소개

2단계 커밋은 여러 노드에 걸친 원자적 트랜잭션 커밋을 달성하는 알고리즘이다. 일부 DB에서는 2PC(2 Phase Commit) 이 내부적으로 사용되고, XA트랜잭션이나 MSA 웹서비스 용 WS- atomic transcation을 통해 애플리케이션에서도 사용할 수 있다.

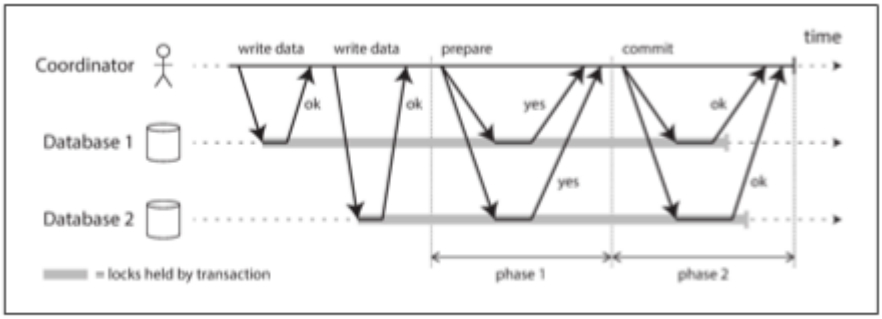


Figure 9-9. A successful execution of two-phase commit (2PC).

단일 노드 트랜잭션에서처럼 하나의 커밋 요청을 하는 대신 2PC의 커밋/어보트 과정은 두 단계로 나뉜다.

2PC는 코디네이터라는 새로운 컴포넌트를 사용한다. 라이브러리나, 프로세스, 서비스 형태를 띌수도 있다.

2PC 트랜잭션은 평상시 처럼 여러 데이터베이스 노드에서 데이터를 읽고 쓰면서 시작한다. 이런 데이터베이스 노드를 트랜잭션의 참여자라고 부른다. 애플리케이션이 커밋할 준비가 되면 코디네이터가 1단계를 시작한다.각 노드에 준비 요청을 보내서 커밋할 수 있는 지 물어본다. 그 이후에 코디네이터가 참여자들의 응답을 추적한다.

- 모든 참여자가 **네**라고 응답하면 코디네이터는 2단계에서 커밋 요청을 보내고 커밋이 실제로 일어난다.
- 참여자 중 누구라도 아니오 라고 응답하면 2단계에서 어보트를 한다.(만장일치가 되어야 한다!)

약속에 관한 시스템

그런데, 준비 요청과 커밋 요청도 여러 이유로 손실될 수있다. 그런 위험이 있는데 2PC가 어떻게 원자성을 보장한다는 걸까? 자세히 알아보자

- 애플리케이션은 분산 트랜잭션을 시작하기 원할 때 코디네이터에 트랜잭션 ID를 요구한다. 이 트랜잭션 ID는 전역적으로 유일하다.
- 애플리케이션은 각 참여자에서 단일 노드 트랜잭션을 시작하고, 해당 트랜잭션에 이전에 받은 ID를 붙인다. 모든 읽기와 쓰기는 이런 단일 노드 트랜잭션중 하나에서 실행된다. 이 단계에서 뭔가 잘못되면 코디네이터나 노드에서 어보트할 수 있다.
- 애플리케이션이 커밋할 준비가 되면 코디네이터는 모든 참여자에게 전역 트랜잭션 ID로 태깅된 준비 요청을 보낸다. 요청 중 실패/타임아웃이 나면 코디네이터가 어보트 요청을 모든 참여자에게 보낸다.
- 참여자는 준비요청을 받으면 분명히 트랜잭션을 커밋할 수 있는지 확인한다. 여기에는 모든 트랜잭션 데이터를 디스크에 쓰는 것과, 충돌, 제약 조건 확인까지 포함된다. 코디네이터에게 yes 을답을 주면서 요청이 있으면 무조건 오류없이 커밋할 것을 보장한다 (어보트할 권리를 포기함)
- 코디네이터는 모든 응답을 받고 트랜잭션을 커밋할 건지 어보트할 건지에 대한 최종 결정을 한다. 코디네이터는 이 결정을 추후 트래킹하기 위해서 디스크에 있는 트랜잭션 로그에 기록해야한다. 이를 **커밋 포인트** 라고 한다.
- 코디네이터의 결정이 디스크에 기록되면 모든 참여자에게 커밋이나 어보트 요청이 전송된다. 이 요청이 실패하거나 타임아웃이 되면 코디네이터는 성공할 때까지 영원히 재시도 해야 한다. 돌아갈 곳이 없다! 도중에 한 참여자가 죽었다면 트랜잭션은 그 참여자가 복구올때 커밋된다.

두가지 포인트가 있는데,

- 참여자는 네를 투표를 할때 나중에 분명 커밋할거라 약속하고 어보트 권리를 포기한다.
- 코디네이터가 한번 결정하면 그 결정은 변경할 수 없다.

이 두가지가 2PC 의 원자성을 보장한다.

코디네이터 장애

위에서 참여자 노드가 장애가 발생할 경우에도 계속 시도한다는 점을 분명히 했다. 그런데 코디네이터에 장애가 생기면 어떻게 되는지는 분명하지 않다.

- 코디네이터가 준비 요청을 보내기 전에 장애가 나면 참여자가 안전하게 트랜잭션을 어보트할 수 있다.

- 하지만 "네" 투표 이후에 장애가 나면 참여자는 기다릴 수 밖에 없다. 이 상태의 참여자 트랜잭션을 **의심스럽다(in doubt) 또는 불확실하다(uncertain)** 고 한다.

2PC 를 완료할 유일한 방법은 코디네이터 복구를 기다리는 방법이다. 이 때문에 코디네이터는 자신의 결정을 디스크에 쓴다.

3단계 커밋

2단계 커밋은 코디네이터 복구를 기다리느라 멈출 수 있다는 사실 때문에 **블로킹** 원자적 커밋 프로토콜이라고 불린다. 이론상으로는 이를 논 블로킹하게 만들 수 있지만 현실에서는 어렵다.

2PC에 대한 대안으로 3PC가 제안됐다. 하지만 3PC의 가정은 지연과 응답시간에 제한이 있는노드다. 기약없는 네트워크 중단과 프로세스 지연이 있는 현실은 3PC에 적합하지 않다. **논블로킹** 원자적 커밋은 **완벽한 장애 감지기(perfect failure detector)**, 즉 노드가 죽었는지 아닌지 구별할 수 있는 신뢰성 있는 메카니즘이 필요하다. 현실에서는 타임아웃이 신뢰성 있는 메카니즘이 아니다.

현실의 분산 트랜잭션

현실의 분산 트랜잭션은 평판이 엇갈린다.

- 중요한 안정성을 달성하는 방법이다.
- 운영상의 문제를 일으키고, 성능을 떨어트리고 그것들이 제공할 수 있는 것들보다 더 약속한다고 비판 받는다.

어떤 분산 트랜잭션 구현은 무거운 성능 손해를 수반한다. MySQL의 분산 트랜잭션은 단일 노드 트랜잭션보다 10배 이상 느리다. 이 비용은 장애 복구를 위해 필요한 디스크 강제 쓰기와 부가적인 네트워크 왕복 시간 때문이다.

그러나 단점이 크다고해서 일축할게 아니라 좀 더 자세히 지켜봐야 한다. 이로부터 배울 수 있는 교훈이 있기 때문이다. 일단 분산 트랜잭션은 두 가지 종류가 있다.

- 데이터 베이스 내부 부분 트랜잭션
    - 복제와 파티셔닝을 사용하는 데이터베이스는 노드사이에 내부 트랜잭션을 지원한다. 이 경우 모든 노드가 동일한 DB 소프트웨어를 사용한다.
- 이종(heterogeneous) 분산 트랜잭션
    - 참여하는 노드가 서로 다른 DB 소프트웨어를 사용한다. 이를 테면 두 가지 서로 다른 벤더의 데이터베이스인 Redis와 MySQL 동시에 쓰기가 있을 수 있겠다. 이런

> 시스템에 걸친 분산 트랜잭션은 시스템의 내부가 완전히 다르더라도 원자적 커밋을 보장해야한다.

정확히 한번 메시지 처리

이종 분산 트랜잭션은 다양한 시스템이 강력하게 통합될 수 있게한다. 예를 들어 메시지 큐의 메시지는 그 메시지를 처리하는 DB 트랜잭션이 커밋에 성공했을 때만 처리된 것으로 확인받을 수 있다.

이런 분산 트랜잭션은 트랜잭션의 영향을 받는 모든 시스템이 동일한 원자적 커밋 프로토콜을 사용할 수 있을 때만 가능하다. 예를 들어 메시지를 처리하는 부수 효과가 이메일 전송이고, 이메일 서버는 2단계 커밋을 지원하지 않는다고 했을때 메시지 처리가 실패하고 재시도되면 이메일이 두번 이상 전송될 수 있다.

XA 트랜잭션

X/Open XA(eXtended Architecture). XA는 PostgresSQL, MySQL, DB2, SQL Server, Oracle, 그리고 ActiveMQ 등의 메시지 브로커에서도 지원한다.

XA는 네트워크 프로토콜이 아니다. 트랜잭션 코디네이터와 연결하는 인터페이스를 제공하는 C API 이다. 다른 언어에도 이 API 가 있다. 자바 EE에서는 XA 트랜잭션이 Java Transaction API (JTA) 를 사용해 구현되며 JTA는 다시 JDBC나 JMS 를 사용하는 시스템에서 지원된다.

트랜잭션 코디네이터는 XA API 를 구현한다. 현실에서는 코디네이터가 단순히 트랜잭션 시작 어플리케이션과 같은 프로세스에 로딩되는 같은 라이브러리다. 참여자를 추적하고 준비 요청을 보내고, 응답을 수집하고 각 트랜잭션에 커밋, 어보트 결정을 추적하기 위해 로컬 디스크에 있는 로그를 사용한다.

의심스러운 상태의 코디네이터, 잠금을 유지하는 것의 문제

의심스러운 상태의 코디네이터가 생기면, 트랜잭션은 보통 로우 수준의 잠금을 획득하므로 문제가 생긴다. 데이터베이스는 트랜잭션이 커밋/어보트할 때까지 이 잠금을 해제할 수 없다. 코디네이터 로그가 어떤 이유로 완전히 손실되면 이 잠금은 영원히, 혹은 관리자가 수동으로 해결할때까지 유지된다.

코디네이터 장애에서 복구하기

이론상으로는 코디네이터가 죽은 후 재시작하면 로그로부터 그 상태를 깨끗하게 복구하고 의심스러운 트랜잭션을 해소해야한다. 그러나 **고아가 된(orphaned)** 트랜잭션이 생길 수 있는데, 즉 어떤 이유인지 그 결과를 결정할 수 없는 트랜잭션이다. (로그 손실됐거나 버그로 오염 됐기 때문에) 이런 트랜잭션은 자동으로 해소가 안되어서 잠금을 유지하고 다른 트랜잭션을 차단하면서 영원히 데이터베이스에 남는다. DB 서버를 재부팅해도 2PC를 제대로 구현했다면 의심스러운 트랜잭션은 잠금이 된다.

여기서 빠져나가는 유일한 방법은 관리자가 수동으로 트랜잭션을 커밋하거나 롤백할지 결정하는 것이다. 이 결정을 할때는 심각한 서비스 중단일 가능성이 높다. 여러 XA 구현에는 참여자가 코디네티어로부터 확정적 결정을 얻지 않고 의심스러운 트랜잭션을 어보트하거나 커밋할지 일방적으로 결정할 수 있는 **경험적 결정(heuristic descision)** 이 있다. 큰 장애 상황에 벗어나고자 할 때만 사용되어야 한다.

분산 트랜잭션의 제약

XA 트랜잭션은 여러 참여 데이터 시스템이 서로 일관성을 유지하게 해주지만, 중요한 운영상 문제도 위와 같이 생긴다. 특히 핵심 구현은 코디네이터 자체가 트랜잭션 결과를 저장할 수 있는 일종의 데이터베이스 여야하고, 따라서 다른 중요 데이터베이스와 동일하게 신경써서 접근해야한다.

- 코디네이터가 복제가 없고 단일 장비에서 실행되면 전체 시스템의 단일 장애점이 된다. 놀랍게도 여러 코디네이터 구현은 기본적으로 고가용성 X 거나 기초적인 복제만 지원한다.
- 코디네이터가 어플리케이션 서버의 일부가 되면 배포의 특성이 바뀌게 된다. Stateful 해진다.
- XA 는 여러 시스템에 걸친 교착 상태를 감지할 수 없다. 그리고 SSI 와 함께 동작하지 않는다.
- 데이터베이스 내부 분산 트랜잭션은 그 제한이 그리 크지 않다. (XA 가 아닌) 그러나 여전히 모든 참여자가 응답해야하므로 장애를 증폭시키는 경향이 있으며 이는 내결함성을 지닌 시스템 구축이라는 목적에 부합하지 않는다.

내결함성을 지닌 합의

서로 공존할 수 없는 연산 중 어떤 것이 승자가 돼야 하는지 결정하는데 합의 알고리즘을 사용할 수 있다.

- 균일한 동의 : 어떤 두 노드도 다르게 결정하지 않는다.

- 무결성 : 어떤 노드도 두번 결정하지 않는다.
- 유효성: 한 노드가 값 v를 결정한다면 v는 어떤 노드에서 제안되었던 값이다.
- 종료: 죽지않은 모든 노드는 결국 어떤 값을 결정한다.

균일한 동의와 무결성 속성은 합의의 핵심 아이디어를 정의한다. 유효성 속성은 뻔한 해결책을 배제한다. 예를 들면, 안되면 무조건 null로 세팅하는 알고리즘이 있을 수 있다. 내결함성이 상관없다면 처음 세개를 만족시키는 것은 쉽다. 그냥 한 노드를 독재자로 하드코딩하면 된다. 그러나 그 노드에 장애가 나면 그 시스템은 어떤 결정도 할 수 없다. 여기서 종료 속성은 내결함성의 아이디어를 형식화 한다. 이 속성은 본질적으로 합의 알고리즘은 그 상태에 머무르기만 할 수 없다는 이야기를 하는 것이다. 다시 말해 진행해야한다. 어떤 노드가 장애가 나도 다른 노드들은 여전히 결정을 내려야한다.

종료 속성은 죽거나 연결할 수 없는 노드가 절반 이하라는 가정이 종속적이다. 그러나 대부분 합의 구현은 과반수 노드가 장애가 나더라도 위 세가지(균일한 동의, 무결성, 유효성) 은 만족하므로 요청을 처리할수 없을 지라도 유효하지 않는 결정을 내리진 않는다.

또 합의 알고리즘은 비잔틴 결함이 없다고 가정한다. (모든 노드는 프로토콜을 올바르게 따르고 있다고 생각함.)

합의 알고리즘과 전체 순서 브로드 캐스트

내결함성을 지닌 합의 알고리즘 중 가장 널리 알려진 것은 뷰 스탬프 복제(Viewstamped replication), paxos, raft, zab 이다. 이 내용을 상세히 다루지는 않는다. 이 알고리즘 중 대다수는 형식적 모델을 직접 사용하지 않는다. 대신 값의 sequence를 정해서 전체 순서 브로드캐스트 알고리즘을 만든다. 이는 앞서 말했듯이 합의로 변환할 수 있다.

전체 순서 브로드캐스트는 모든 노드에게 메시지가 정확히 한번, 같은 순서로 전달된다. 이는 합의를 몇회하는 것과 동일하다.

- 합의의 동의 속성때문에 모든 노드는 같은 메시지를 같은 순서로 전달하도록 결정
- 무결성 때문에 메시지는 중복 되지 않는다.
- 유효성 속성때문에 메시지는 오염되지않고 난데 없이 조작되지 안흔다.
- 종료 속성때문에 메시지는 손실되지 않는다.

잠깐 단일 리더 복제를 생각해보자. 리더가 없어지면 새로운 리더를 선출해야한다. 합의는 전체 순서 브로드캐스트와 같고, 전체 순서 브로드캐스트는 단일리더와 같고, 단일리더 복제는 리더가 필요하고..

리더를 새로 선출하려면 리더가 필요한것같다. 합의를 하려면 합의를 해야한다. **이 난제를 어떻게 해결할 수 있을까?**

에포크 번호 붙이기와 정족수

지금까지의 합의 프로토콜은 모두 내부적으로 어떤 형태로든 리더를 사용하지만 리더가 유일하다고 보장하지는 않는다. 대신 약한 보장을 한다. 에포크 번호를 정의하고 각 에포크 내에서는 리더가 유일하다고 보장한다.

리더가 죽었다고 생각될때마다 투표가 시작된다. 이 선출은 에포크 번호를 증가시키며 따라서 에포크 번호는 전체 순서가 있고 단조 증가한다. 다른 에포크에 있는 리더 사이에 충돌이 있으면 번호가 큰 쪽이 이긴다.

리더는 노드의 정족수로부터 투표를 받는다. 노드는 에포크 번호가 더 높은 리더를 알지 못할 때만 제안에 찬성하는 투표를 한다.

따라서 두번의 투표가 있다.

- 한번은 리더 선출
- 두번째는 리더의 제안에 투표

중요한 것은 두번 투표하는 정족수가 최소 한명은 겹쳐야한다. 제안에 참여하는 노드 중 최소 하나는 가장 최근의 리더 선출에 참여했어야 리더는 다른 (에포크가 높은) 리더가 없다고 확신하고 안전하게 제안된 값을 결정할 수 있다.

겉보기에는 2PC와 비슷해보인다. 가장 큰 차이는 2PC에서 코디네이터는 선출되지 않는다는 것과 2PC는 모든 참여자로부터 "네" 가 필요하지만 내결함성을 지닌 합의 알고리즘은 과반수로부터만 투표를 받으면 된다는 것이다.

합의의 제약

합의 알고리즘은 분산시스템의 커다란 발전이다.

- 구체적인 안전성 속성을 가져와준다.(동의, 무결성, 유효성)
- 그럼에도 내결함성을 유지한다(노드의 과반수가 동작하는 한)
- 전체 순서 브로드캐스트를 제공하고 따라서 내결함성 있는 방식으로 선형성 원자적 연산을 구현할 수 있다.

그럼에도 모든 곳에 쓰이지않는다. 비용이 따르기 때문이다.

- 노드가 제안에 투표하는 과정은 일종의 동기식 복제다. (성능에 손실)

- 합의 시스템은 엄격한 과반수가 동작하기를 요구한다.
- 대부분 합의 알고리즘은 투표에 참여하는 노드가 고정되어있다고 가정한다. 이는 클러스터에 그냥 노드를 추가하거나 제거할 수 없다는 뜻이다.
- 합의 시스템은 장애노드 감지를 대개 타임아웃에 의존한다. 네트워크 변동이 심한 환경에서 리더 선출이 빈번하게 생길 수 있는 가능성이 있다.
- 네트워크 문제에 민감하다.

멤버십과 코디네이션 서비스

주키퍼나 etcd같은 프로젝트는 종종 분산 key-value 저장소 혹은 코디네이션과 설정 서비스라고 설명된다. 이게 데이터베이스라면 이것들은 왜 합의 알고리즘에 쓰이는걸까?

주키퍼는 보통 다른 프로젝트를 통해 간접적으로 의존하게 된다. e.g.) HBase, Hadoop Yarn, OpenStack Nova, kafka 등.

주키퍼와 etcd는 완전히 메모리 안에 들어올 수 있는 작은 데이터를 보관하도록 설계되었다. 이 데이터는 내결함성을 지닌 전체 순서 브로드캐스트 알고리즘을 사용해 모든 노드에 걸쳐 복제된다.

주키퍼는 구글의 Chubby 잠금 서비스를 모델로 삼아 전체 순서 브로드캐스트(그리고 합의도)를 구현할 뿐 아니라 **다른 유용한 기능**도 구현한다.

- **선형적 원자적 연산**
  - compare-and-set을 이용해 잠금을 구현할 수 있다. 분산 잠금은 보통 제한 시간이 있는 잠금, 즉 임차권(lease)으로 구현된다.
- **연산의 전체 순서화**
  - 펜싱토큰. 주키퍼는 모든 연산에 전체 순서를 정하고 트랜잭션 ID와 버전 번호를 할당해 달성한다.
- **장애 감지**
  - 클라이언트는 주키퍼 서버에 수명이 긴 세션을 유지하고 클라이언트와 서버는 주기적으로 하트비트를 교환해서 다른쪽이 살아있는지 확인한다. 세션 타임아웃보다 긴 시간동안 하트비트가 멈추면 주키퍼는 세션이 죽었다고 선언한다
- **변경 알림**
  - 클라이언트는 다른 클라이언트가 생성한 잠금과 값을 읽을 수 있을 뿐만 아니라 거기에 변경이 있는지 감시할 수도 있다.

작업을 노드에 할당하기

주키퍼/처비 모델이 잘 동작하는 예 1

- 여러 개의 프로세스나 서비스가 있고, 그 중 하나가 리더나 주 구성요소로 선택돼야할 때다. 단일리더 데이터베이스, 작업 스케줄러나 비슷한 상태저장시스템에도 유용하다

또다른 예 2

- 파티셔닝된 자원이 있고 어떤 파티션을 어느 노드에 할당해야할지 결정하는 경우.
- 파티셔닝 재균형화

이 종류의 작업은 주키퍼에서 원자적연산, 단명 노드, 알림을 잘 사용하면 사람의 개입 없이 결함으로부터 자동 복구하면서 잘 돌아간다. 애플리케이션은 결국 수천대의 노드로 늘어날 수도 있다. 매우 많은 노드에서 과반수 투표를 하는 것은 비효율적이므로, 주키퍼는 노드들을 코디네이트하는 작업의 일부를 외부 서비스의 위탁 하는 방법을 제공한다.

서비스 디스커버리

주키퍼, etcd, 콘술은 서비스 찾기의 용도로 자주 사용된다. 예를 들어, 특정 서비스에 연결하려면 어떤 IP로 접속 해야 하는지 질의할 떄 사용될 수 있다.

그런데 서비스 찾기가 실제로 합의가 필요한지는 분명해보이지 않는다. DNS는 서비스 이름으로 IP 주소를 찾는 전통적인 방법이고, 좋은 성능을 위해 다층 캐시를 이용한다. 이러면 분명히 기능이 선형적이지 않지만, 질의 결과가 뒤처지더라도 문제로 생각되지 않는다.

멤버십 서비스

주키퍼와 유사 프로젝트들은 오랜 멤버십 서비스 연구의 일부다. 멤버십 서비스는 클러스터에서 어떤 노드가 활성화된 멤버인지 결정한다. 장애 감지를 합의와 연결하면 실제 상태와는 사실 무관하게어떤 노드가 살아있다고 여겨지고, 어떤게 죽었다고 여겨져야하는지 동의할 수 있다.

EX) 카프카 멤버십

같이 읽어보면 좋은 글

- XA 트랜잭션에 대하여: https://myinfrabox.tistory.com/116