CH006.

키-값 저장소 설계

- 너무 어려운 주제
- 교재 112~114p 내용 제외

키-값 저장소란?

- Non-relational Database
- 유일한 **키**로 값에 접근하는 **키-값** 쌍을 사용함

```
// key는 문자열 or 해시값
key: "last_logged_in_at" or "253DDEC4"
```

인터페이스

- put(key, value): 키-값 쌍을 저장
- get(key): 인자로 주어진 키로 값을 조회

단일 키-값 저장소의 문제점

- 가장 직관적인 방법
  - **메모리에 해시 테이블로 키-값 쌍을 저장함**
- 다음과 같은 기법을 활용
  - 데이터 압축
  - 자주 쓰이는 데이터에 대한 캐싱

단일 키-값 저장소의 문제점

- 많은 데이터를 저장하려면?
  - 즉, 모든 데이터를 독립적인 메모리 공간에 저장하는게 분산 환경에 적합한가?

분산 키-값 저장소로 해결

- 키-값 쌍을 여러 서버에 분산 저장함

분산 키-값 저장소 - CAP Theorem

- 데이터 일관성(Consistency)
- 데이터 가용성(Availability) ➡️ **"모든 장애에 강해야 함"**
- 파티션 감내(Partition tolerance) ↗️
- 위 세 가지를 모두 만족할 수는 없다는 이론

분산 키-값 저장소 - [C]AP

- 어떤 노드냐 상관없이 **언제나 같은 데이터**를 조회

분산 키-값 저장소 - C[A]P

- 어떤 노드에 **장애가 있어도 항상 응답**을 보장

분산 키-값 저장소 - CA[P]

- 두 노드 간에 **통신 장애**가 발생함
  - 이를 파티션이라 함
- 감내란 이 둘 사이에 **파티션이 발생해도 전체 시스템은 동작해야 함**을 의미

분산 키-값 저장소 - CAP Theorem

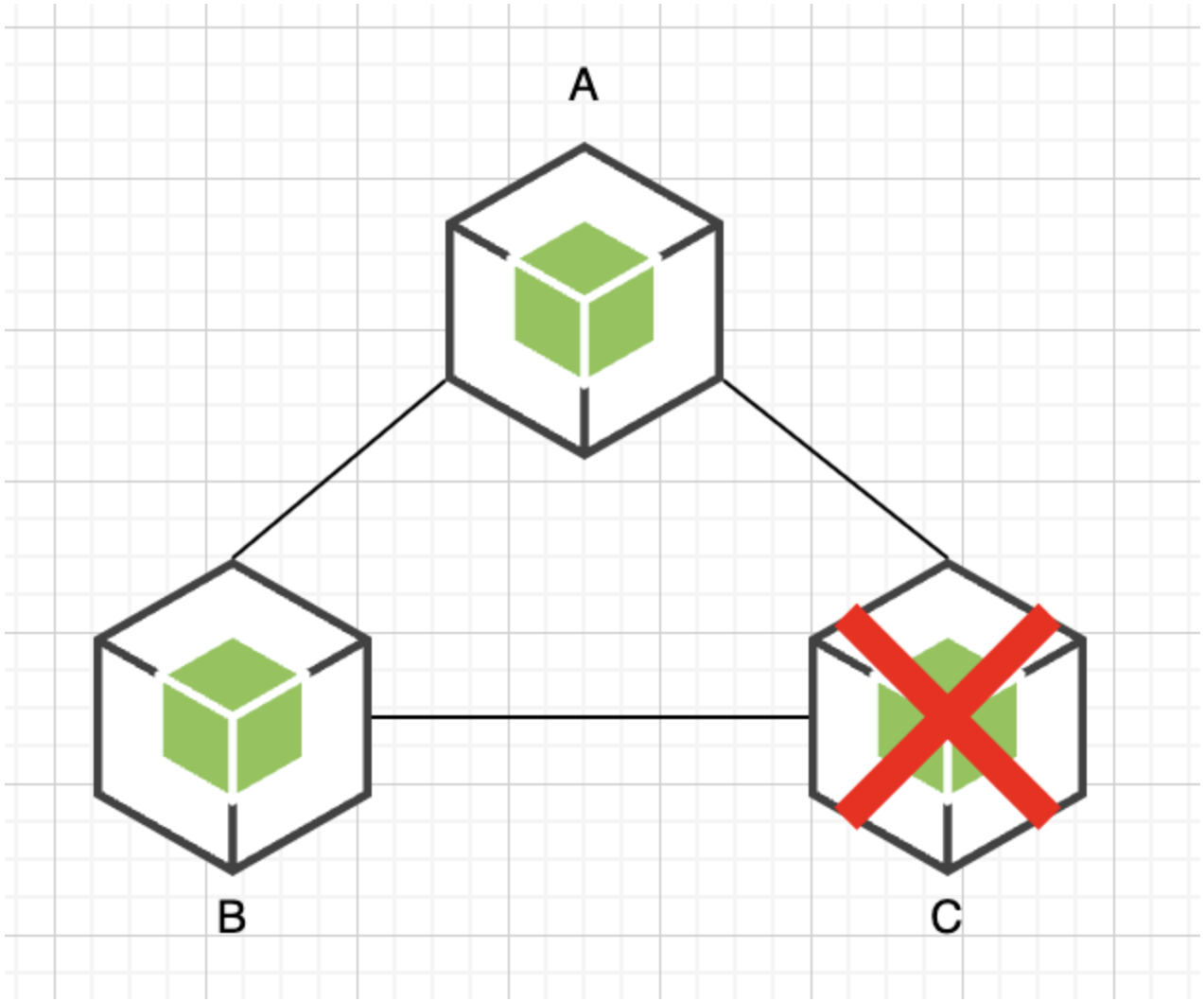
- C, A, P 중에 하나는 반드시 죽는다.

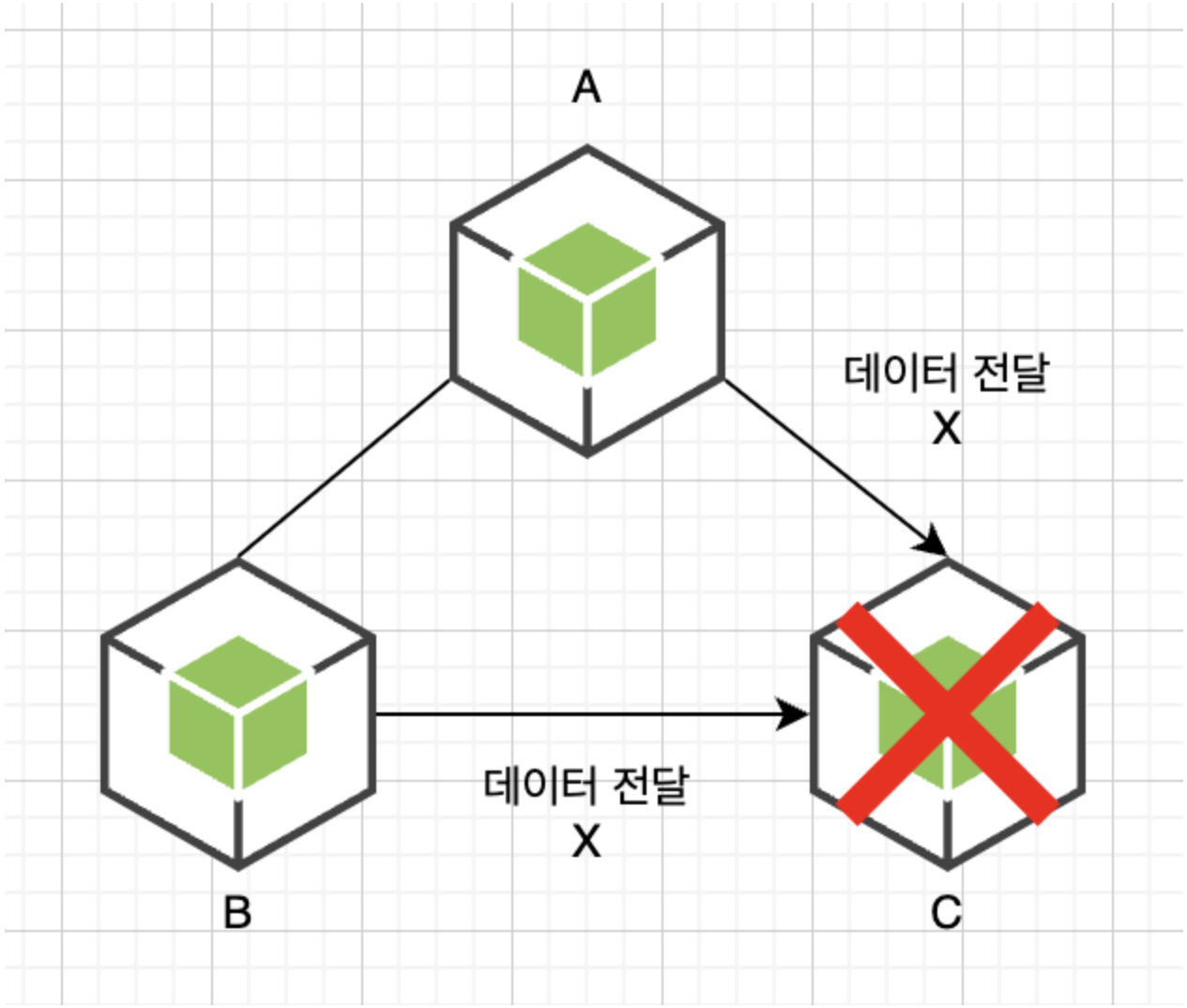
분산 키-값 저장소 - CAP Theorem

- 근데 사실 P는 무조건 살아야 한다.

분산 키-값 저장소 - 왜 P만 살려야 할까?

- 통상 네트워크 장애는 피할 수 없음
- 따라서 모든 분산 시스템은 P를 만족해야 함
- 그러므로 CA 시스템이란 있을 수 없음


A
B
C

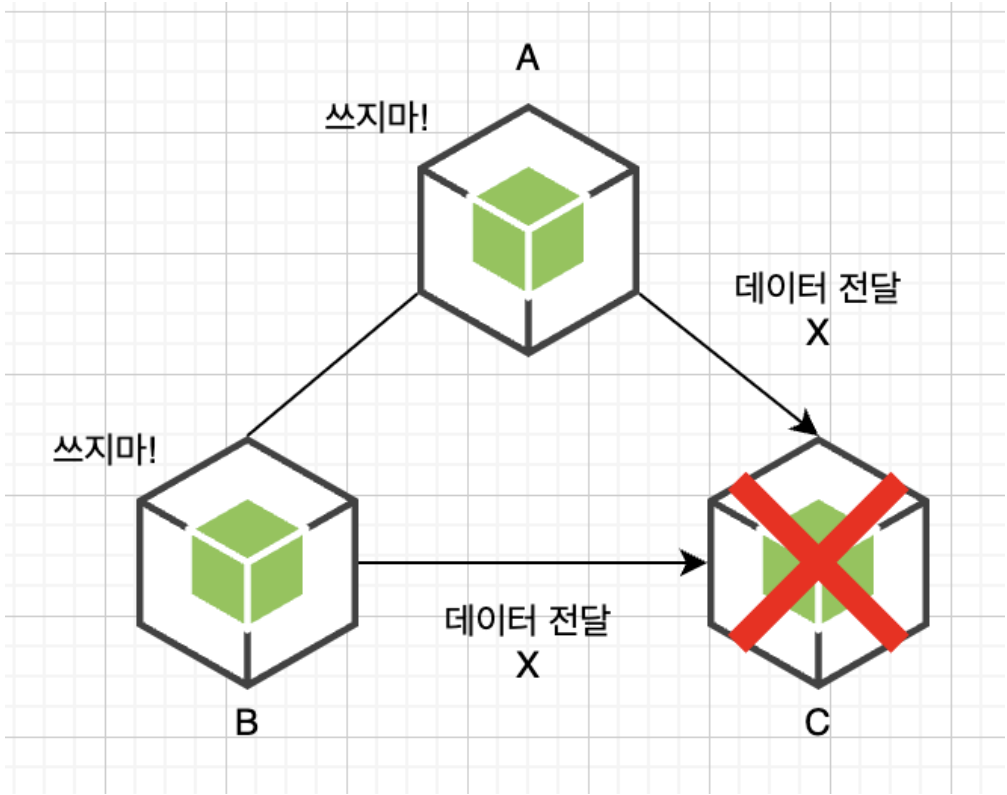


- `C`는 오래된 사본을 갖고 있게됨

C(일관성)을 선택한다면?

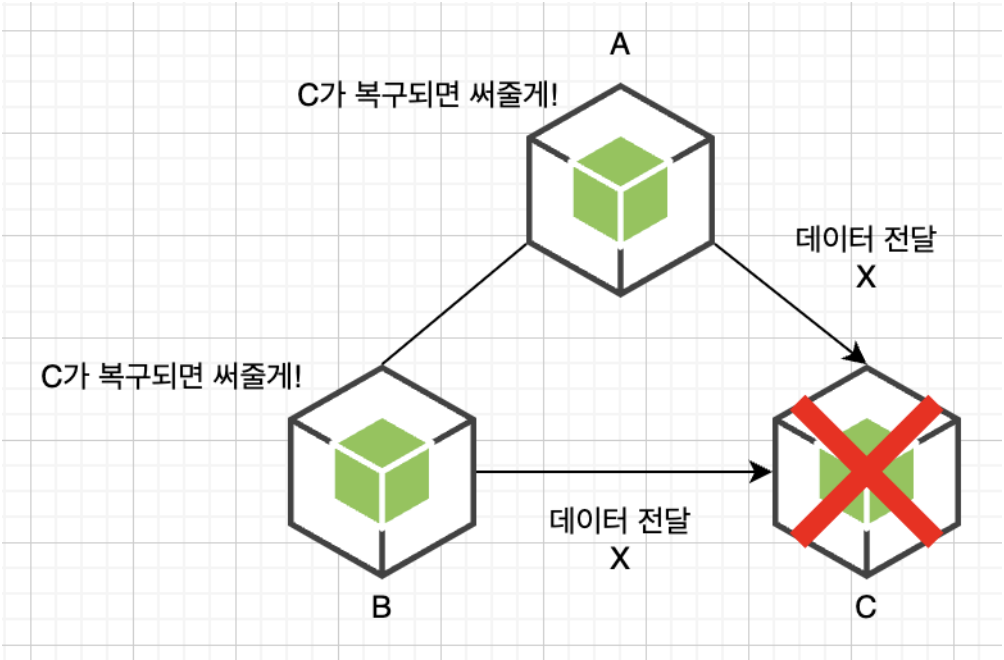
- 은행 계좌 시스템 시나리오에 적합
- **계좌**는 모든 노드에서 **항상 동일한 결과를 반환**해야만 함
- 따라서, `C`가 복구될 때 까지 기다려야만 함

  ➡️ "계속 오류를 반환시킴"
- `A`와 `B`의 **쓰기를 중단**시킴



A(가용성)을 선택한다면?

- 최신 데이터를 보장하지 않지만 **계속 응답** 받을 수 있음
- 따라서, `C` 가 복구되지 않아도 **읽기 허용**
- `A` 와 `B` 의 **쓰기도 허용**시킴
  ➡️ 나중에 `C` 가 복구되면 최신 데이터 전송



키-값 저장소 구현

- 구현을 위한 핵심 컴포넌트들이 있음
- 너무 어려워서 문제(솔직히 벼락치기 함 👃😅)

데이터 파티션

데이터 파티션

- 데이터를 한 곳에 모두 넣는 것은 불가능
  - 데이터를 저장하는 매체의 물리적 용량 한계
  - 정보를 저장하고 처리하는 과정에서 성능 저하
  - 모든 데이터를 압축할 수 없음 ➡️ **비둘기집 원리**

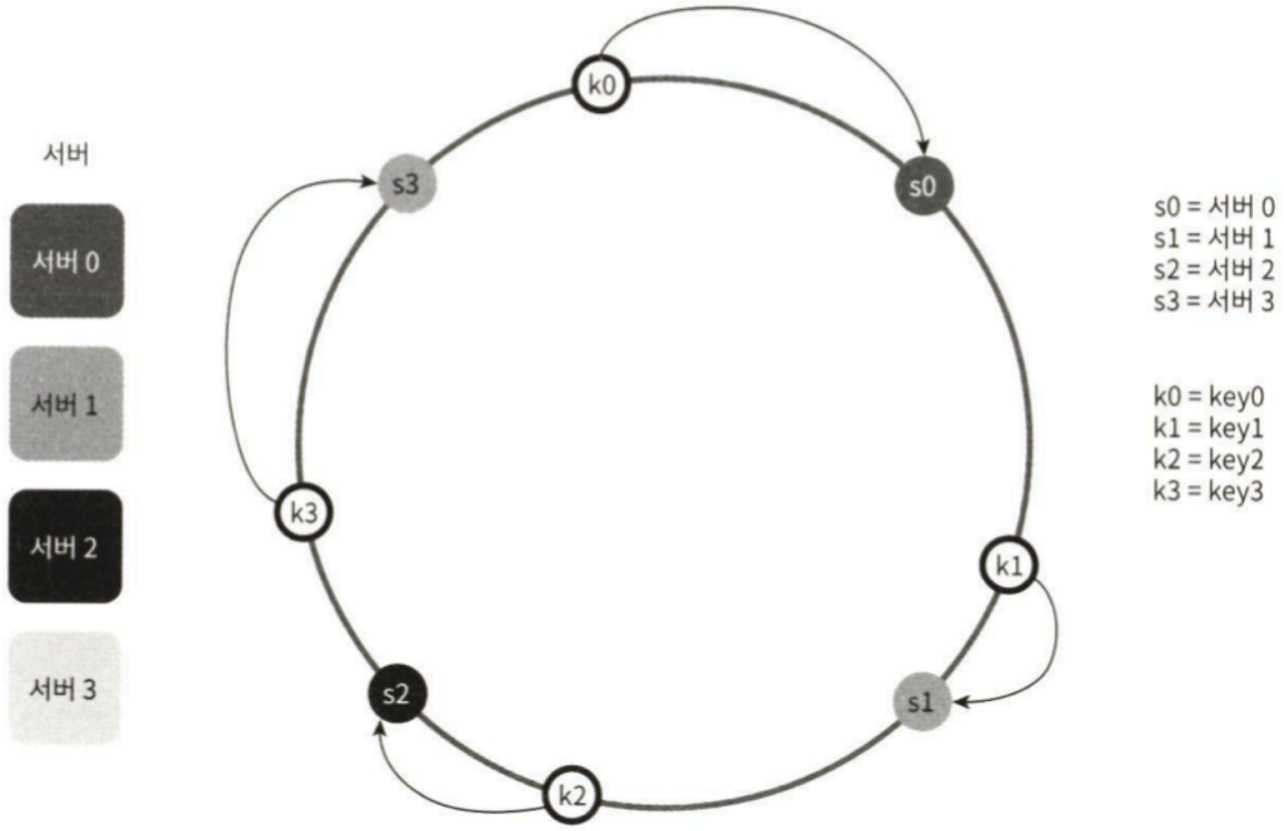
데이터 파티션

따라서...

- 데이터를 여러 서버에 분산 저장해야 함
- 노드가 추가되거나 삭제될 때 데이터 이동을 최소화해야 함

데이터 파이션

이런 문제를 해결하기 위해 **안정 해시**가 필요함 ➡️ 5장 참고



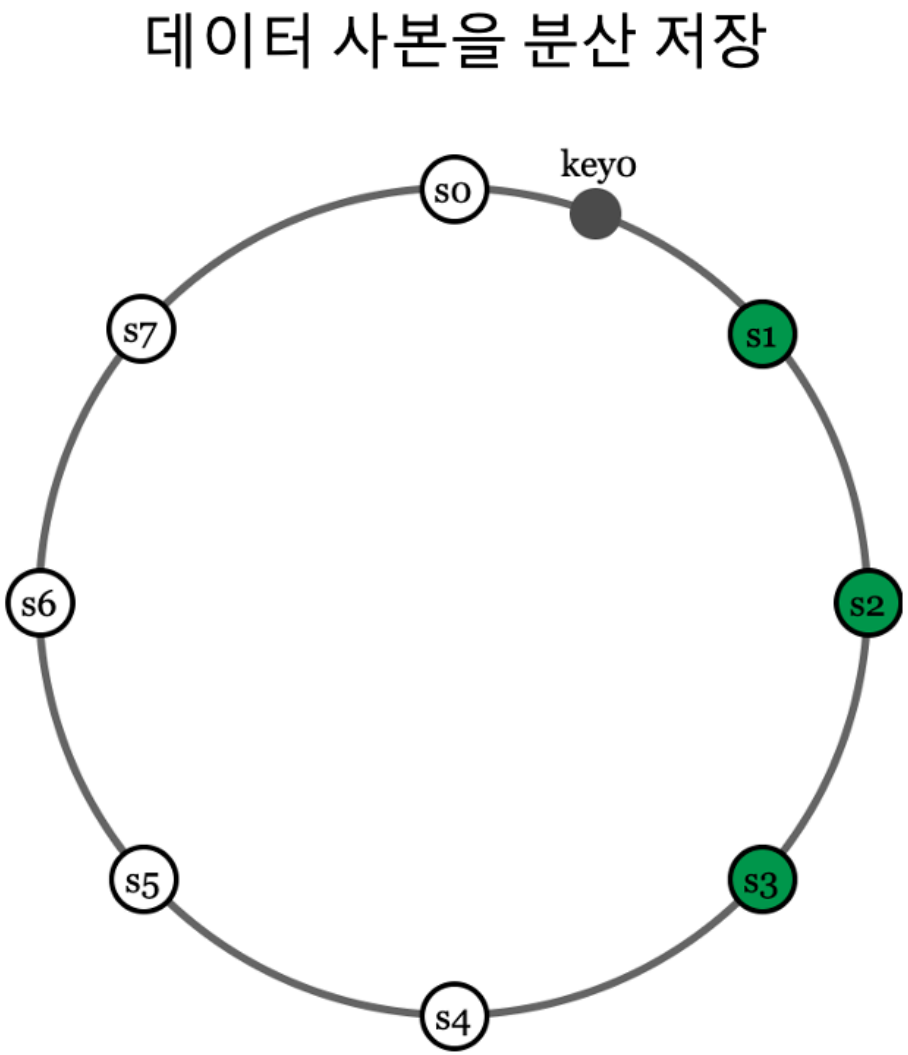
데이터 다중화

데이터 다중화

- 다중화란 레플리케이션(Replication)
- 즉, **데이터의 복제**를 의미함
- 높은 가용성과 안정성 확보

데이터 다중화

- `key0` 이 시계 방향으로 순회해 만나는 `N` 개 서버에 데이터 사본을 저장
- `N=3` 이면 `s1` ~ `s3` 에 저장



데이터 다중화

- 만약에 가상 노드를 사용한다면?
- 실제 대응되는 물리 서버가 `N` 개 보다 작아지는 상황이 발생 가능
- 즉, 같은 데이터 센터에 속한 노드가 어떤 장애를 겪게 되면
  - 특정 서버가 중복되어 선택될 수 있음 ➡️ 한 곳에 데이터가 몰림
  - **따라서, 물리적으로 다른 센터의 서버에 보관할 필요가 있음**

데이터 일관성

데이터 일관성

- **다중화된 데이터를 동기화**해 일관성을 보장해야 함을 의미
- Quorum Consensus 프로토콜 사용
  ➡️ 이를 위한 **중재자**가 필요함

데이터 일관성

- 클라이언트 요청을 중재자가 처리함
- `N` : 사본 서버 개수
- `W` : 이 개수만큼의 서버로부터 쓰기가 성공했음을 응답 받기
- `R` : 이 개수만큼의 서버로부터 읽기가 성공했음을 응답 받기

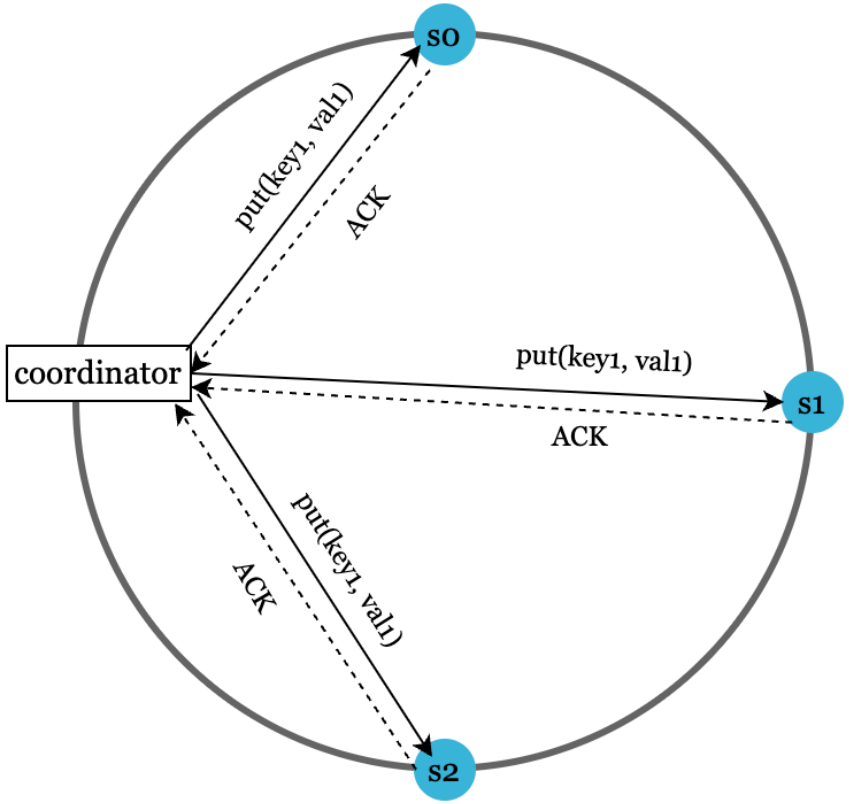


Figure 6 (ACK = acknowledgement)

데이터 일관성

- 빠른 읽기 연산 ➡️ `R=1`, `W=N`
- 빠른 쓰기 연산 ➡️ `W=1`, `R=N`
- 강한 일관성 ➡️ `N < W + R` (`N=3`, `W=R=2`)
- 일관성 보장 안됨 ➡️ `W + R <= N`

데이터 일관성 - 강한 일관성 문제

- 모든 읽기는 최신 데이터를 반영/반환함

  ➡️ 절대로 낡은 데이터를 볼 수 없음

- 생각해 보자
  - 이는 모든 노드가 합의(Consensus)가 필요함을 의미
  - 그 결과 **응답시간 지연**이 필연적으로 발생

    ➡️ 분산 환경에서 성능 저하

데이터 일관성 - 최종 일관성

- **Eventual Consistency**
- 약한 일관성 모델의 한 종류
- 갱신해야 할 데이터가 결국에는 모든 서버에 반영됨을 의미

  ➡️ **동기화**

데이터 일관성 - 최종 일관성 문제

> "쓰기 연산이 병렬적으로 발생하면 시스템에 저장된 값의 일관성이 깨어질 수 있는데, 이 문제는 클라이언트가 해결해야 한다."
> ~ Alex Xu, 가상 면접 사례로 배우는 대규모 시스템 설계 기초, 101p. ~

데이터 일관성 - 최종 일관성 문제

이게 왜 일관성이 깨진다는 걸까?

⬇️ 쉬운 예제

```
철수가 "로봇 장난감 5개 입고"를 메모했을 때, 영희와 민수의 메모장엔 아직 반영이 안된 상황
나중에 시간이 지나야 모든 메모장이 같아짐

- 철수: "로봇 장난감 5개 남음" 기록
- 동시에 영희: "로봇 장난감 3개 팜" 기록
- 동시에 민수: "로봇 장난감 4개 팜" 기록

이때 각자의 메모장은 다른 숫자를 갖게 되고, 실제로 몇 개가 남았는지 모르게 됨
이처럼 분산 환경에서는 동시에 여러 변경이 일어나기에 일관성이 깨짐
```

데이터 일관성 - 최종 일관성 문제

이게 왜 일관성이 깨진다는 걸까?

⬇️ 예제

```
분산 환경에서 사본 서버는 일시적으로 다른 값을 갖게될 수 있음
읽기 연산은 쓰기 연산의 최신 값을 즉시 반영하지 못함
시간이 지나야 결국(Eventually) 모든 노드가 동일함

- 데이터베이스에서 100이란 값을 A와 B가 동시에 읽음
- A는 +5를 하고, B는 -3을 함
- 최종값이 105가 될지, 97이 될지 시스템은 보장할 수 없음
- 실제로 105나 97 모두 틀린 값 (정확히 102여야 하나)

따라서 일관성은 깨짐
```

데이터 일관성 - 최종 일관성 문제

그럼 이걸 왜 클라이언트가 해결해야 할까?

⬇️ 데이터베이스가 이걸 해결한다고 해보자

데이터베이스가 처리하면 100% 정확하겠지만, 그만큼 느려지고 시스템은 복잡해짐
애초에 리얼 월드에서 데이터베이스는 가용성이 아니라 일관성을 위한 시스템 ➡️ 설계 철학의 문제
안그래도 가용성이 부족한데 일관성을 위해 이를 더 감소시키면? ➡️ 노드간 동기화 빡세짐
이는 성능 저하와 구현 복잡도로 이어짐

따라서 비즈니스 로직을 알고 있는 클라이언트가 상황에 맞게 충돌 해결 전략을 구현해야 함
➡️ 그러면 데이터베이스는 기본적인 저장/조회에 집중 가능

데이터 버저닝

- 최종 일관성 문제를 해결하기 위한 기법
- **Version Vector**라고도 함
  - 데이터를 변경할 때마다 해당 데이터의 새 버전을 만듬 ➡️ 이는 변경 불가능
  - Vector clock으로 [서버, 버전]의 순서쌍을 데이터에 매단 것
  - 어떤 버전이 선행인지, 후행인지, 다른 버전과 충돌인지 판별함

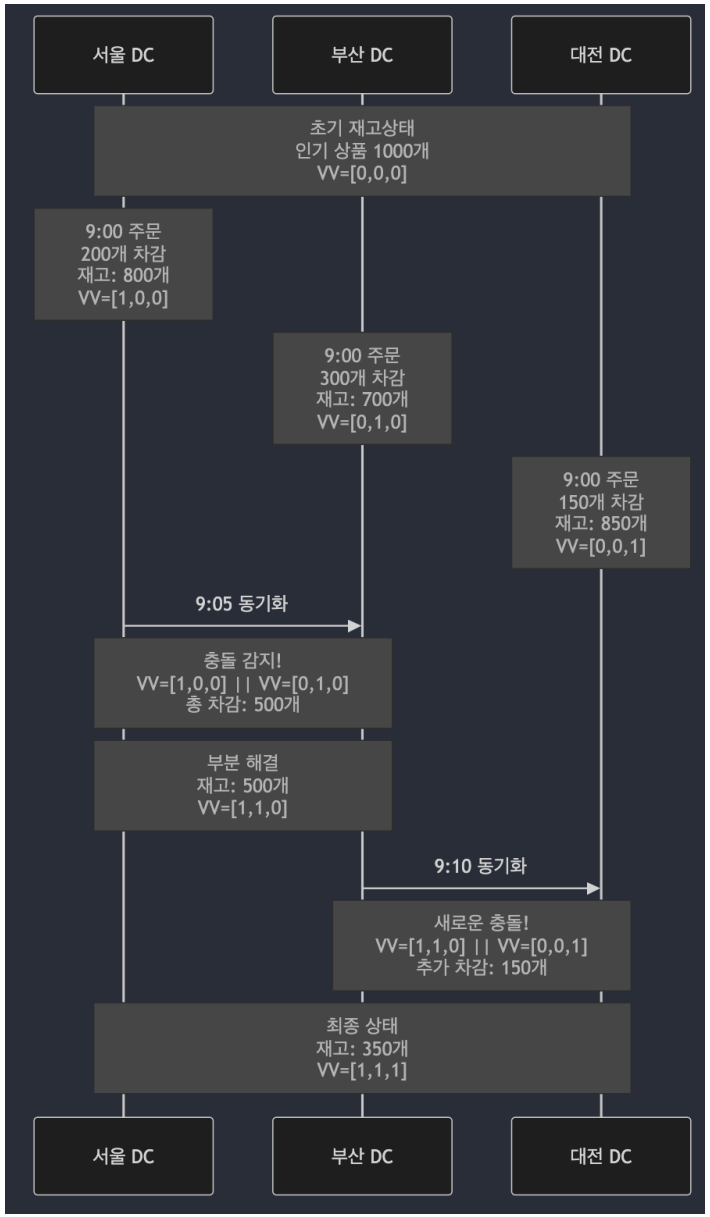
데이터 버저닝

데이터 버저닝은 아래와 같은 연산을 함

- [서버, 버전] 쌍이 있으면 버전을 증가시킴
  ➡️ [`S_i` , `V_i`] is not null `V_i + 1`
- 없으면 새 항목을 만듬
  ➡️ `new` [`S_i` , `1`]

데이터 버저닝 - 기본 원칙

- 각 노드는 자신의 변경사항에 대해서만 버전 카운터를 증가시킴
- Vector 비교 시, 모든 요소가 크거나 같고 적어도 하나는 **반드시** 커야 "더 최신" 이라 함
- 그렇지 않은 경우 **충돌** 관계임


서울 DC
부산 DC
대전 DC
초기 재고상태
인기 상품 1000개
VV=[0,0,0]
9:00 주문
200개 차감
재고: 800개
VV=[1,0,0]
9:00 주문
300개 차감
재고: 700개
VV=[0,1,0]
9:00 주문
150개 차감
재고: 850개
VV=[0,0,1]
9:05 동기화
충돌 감지!
VV=[1,0,0] || VV=[0,1,0]
총 차감: 500개
부분 해결
재고: 500개
VV=[1,1,0]
9:10 동기화
새로운 충돌!
VV=[1,1,0] || VV=[0,0,1]
추가 차감: 150개
최종 상태
재고: 350개
VV=[1,1,1]
서울 DC
부산 DC
대전 DC

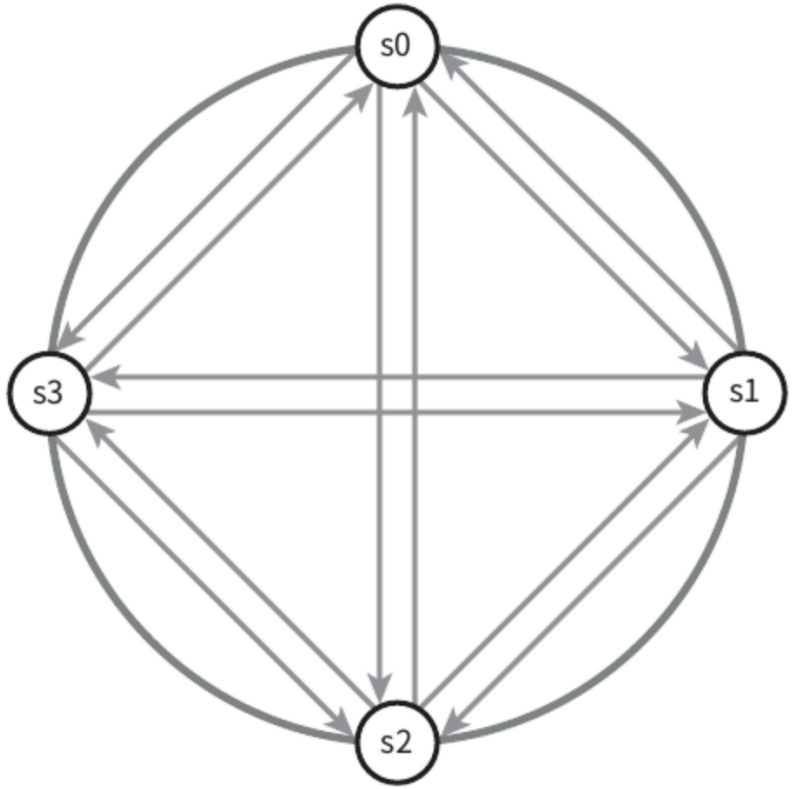
데이터 버저닝 - 문제점

- 충돌을 해결하는 로직이 클라이언트에 있어야 함
- 동시 요청 상황에서 누가 선후 관계인지 파악하기 어려움
  ➡️ 1번 공지사항을 A와 B 어드민이 동시에 수정하면 먼저 반영되는 건?

이제 장애 감지와 처리를 배우자

장애 감지 - 멀티 캐스팅

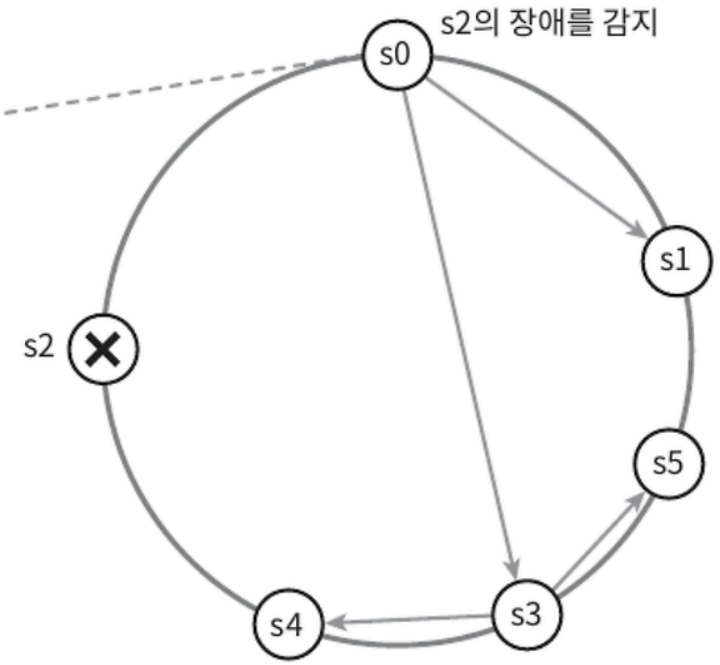
- 각 노드가 멀티캐스트로 하트비트 메시지 전송
- 다른 노드들이 특정 노드의 하트비트를 수신하지 못하면 장애로 판단
- 예를 들어, `D=2` 라고 이상의 노드가 수신 못하면 과반수로 특정 노드는 장애라 판단



장애 감지 - 가십 프로토콜

s0's membership list

Member ID	Heartbeat counter	Time
0	10232	12:00:01
1	10224	12:00:10
2	9908	11:58:02
3	10237	12:00:20
4	10234	12:00:34



- 지정 시간동안 카운터가 갱신되지 않으면 해당 멤버는 장애 상태로 판단

장애 처리 - 느슨한 정족수

- 가십 프로토콜은 특정 시간대에 장애를 탐지함
  ➡️ 만약에 특정 시간대에 장애를 탐지하지 못하면?
- 장애가 있는 노드인데 장애가 없다고 판단하기 때문에 가용성을 보장하지 못함

장애 처리 - 느슨한 정족수

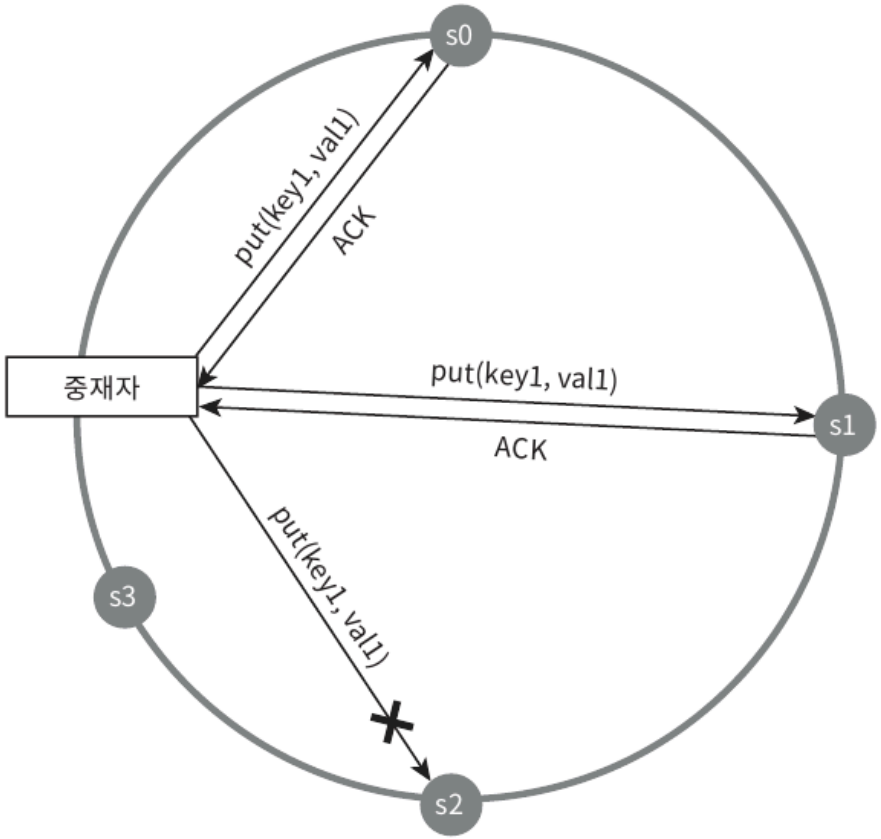
- 엄격한 정족수는 읽기/쓰기 연산을 금지해 일관성을 높임
- 느산한 정족수는 조건을 완화해 가용성을 높임

장애 처리 - 느슨한 정족수

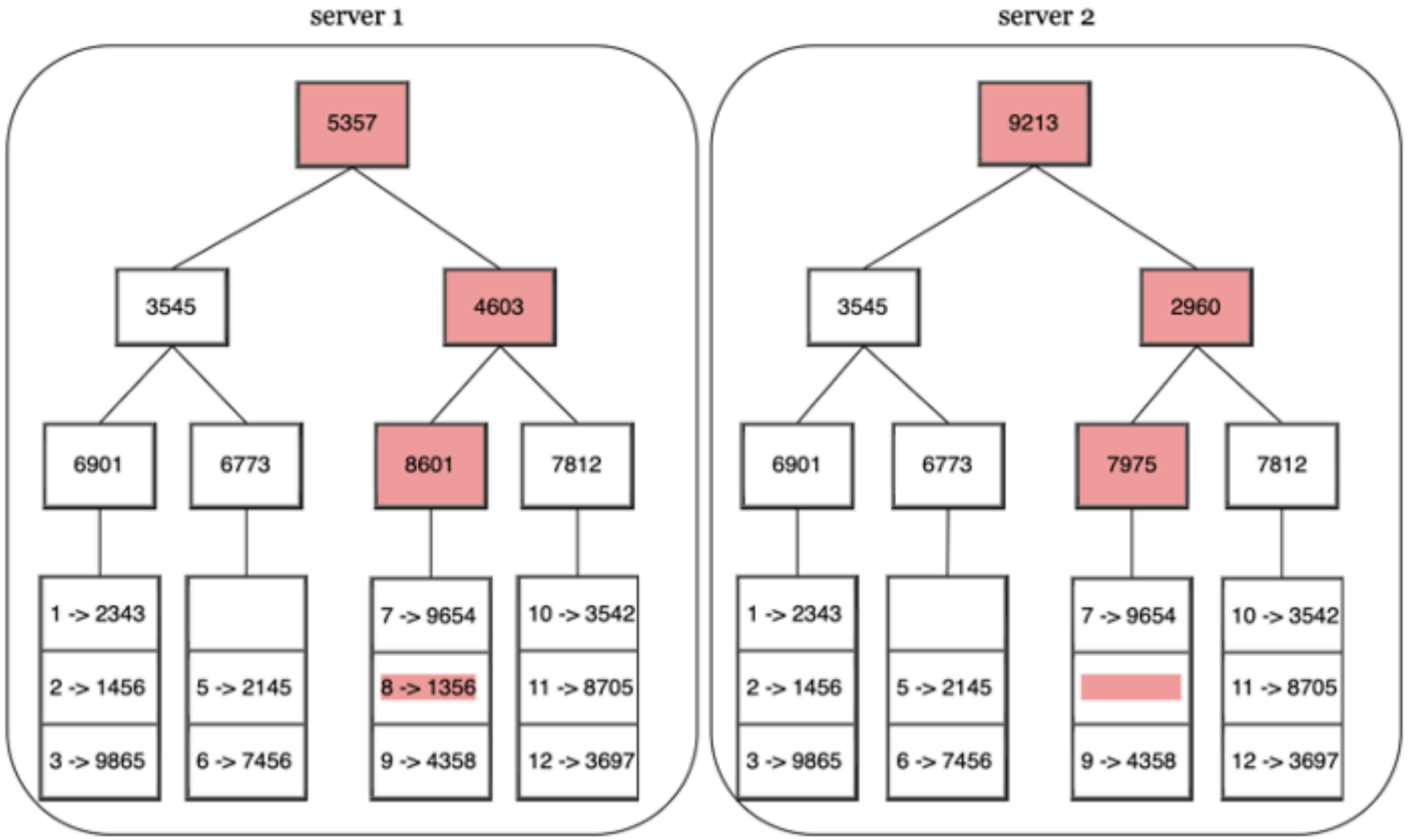
- 쓰기 서버 `W` 개
- 읽기 서버 `R` 개
- 위 두 서버를 해시 링에서 고름
- 그리고 장애 상태 서버는 무시함

장애 처리 - 느슨한 정족수 + 단서 후 임시 위탁

- 장애 상태 서버로 가는 요청을 다른 서버가 맡아서 처리
- 그동안 발생한 변경사항은 장애 서버가 복구되었을 때 일괄 반영함
- 이를 위해 임시 서버에 자신이 수행한 작업에 대한 힌트(메타데이터)를 저장함



영구 장애 처리 - 머클 트리


server 1
5357
3545
4603
6901
6773
8601
7812
1 -> 2343
2 -> 1456
3 -> 9865
5 -> 2145
6 -> 7456
7 -> 9654
8 -> 1356
9 -> 4358
10 -> 3542
11 -> 8705
12 -> 3697
server 2
9213
3545
2960
6901
6773
7975
7812
1 -> 2343
2 -> 1456
3 -> 9865
5 -> 2145
6 -> 7456
7 -> 9654
9 -> 4358
10 -> 3542
11 -> 8705
12 -> 3697

영구 장애 처리 - 머클 트리

- 사본 간에 데이터를 검증할 때, 대용량 데이터라면?
  ➡️ 엄청나게 많은 데이터를 비교해야 해 성능저하 유발
- 해시 값으로 비교하면?
  ➡️ 데이터 총량과 무관하게 해시 값만 비교해 빠르게 넘어갈 수 있음

개인적인 생각

- 블록체인도 데이터 일관성과 가용성에서 줄다리기를 많이 하고 있음 (블록체인 트릴레마)
- 이더리움 만든 천재 개발자 부테린도 해결 못한 굉장히 어려운 주제라고 생각함

그래서 12월 29일에 한 번 더 한다.

끗.