키-값 저장소 설계

개관

- 키-값 저장소 == 키-값 데이터 베이스 (비 관계형 데이터 베이스)
- 키-값의 연관 관계를 키-값 쌍이라고 부르며, 키는 유일해야함.
    - 일반 텍스트 혹은 해시로 이루어져있음.
        - 텍스트 키 : “last_logeed_in_at”
        - 해시 키 : 253DDEC4
- 값은 문자열, 리스트, 객체 등등 어느 것이든 될 수 있음.
- 보통 DynamoDB, memcached, Redis를 많이 사용

키	값
145	john
146	bob
147	julia

- 이번 장에서는 put (저장) , get (조회) 를 설계 해 볼 예정

설계 범위

- 키-값의 크기는 10KB 이하
- 큰 데이터를 저장할 수 있어야 한다
- 높은 가용성을 제공해야 한다
    - 즉, 장애가 있더라도 빠르게 응답해야함
- 높은 규모 확장성을 제공해야 한다
    - 트래픽 양에 따라 자동적으로 서버의 증설 및 삭제가 이루어진다
- 데이터 일관성 수준은 조정이 가능해야한다
- 응답 지연시간이 짧아야한다

단일 서버 키-값 저장소

- 키-값 쌍을 전부 메모리에 해시 테이블로 저장
- 빠른 속도를 보장하지만, 모든 데이터를 메모리에 두는 것이 불가능하다는 약점 존재
    - 데이터 압축 및 자주 쓰는 데이터만 메모리에 두고 나머지는 디스크에 저장하는 방식으로 약점 해결

⇒ 이렇게 개선해도, 한 대의 서버만으로 부족한 경우가 발생. 이 때 분산 키-값 저장소를 만들어야 함.

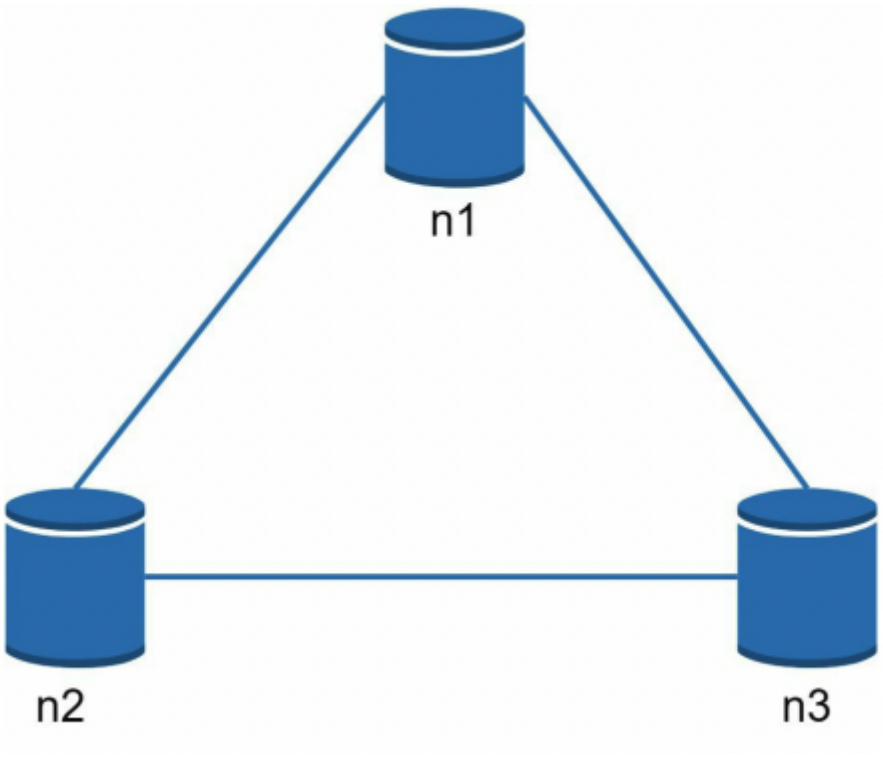
분산 키-값 저장소

- CAP 정리
    - Consistency, Availability, Partition tolerance 를 동시에 만족하는 분산 시스템을 설계하는 것은 불가능.
    - Consistency
        - 분산 시스템에 접속하는 모든 클라이언트는 어떤 노드에 접속했느냐에 관계 없이 언제나 같은 데이터를 보게 되어야 한다.

- Availability
  - 분산 시스템에 접속하는 클라이언트는 일부 노드에 장애가 발생하더라도 항상 응답을 받을 수 있어야 한다
- Partition tolerance
  - 파티션은 두 노드 사이에 통신 장애가 발생하였음을 의미한다. 파티션 감내는 네트워크에 파티션이 생기더라도 시스템은 계속 동작하여야 한다는 것을 뜻한다
- CP, AP, CA 저장소로 나뉘며 보통 하나의 특징을 희생하고, 다른 두 가지를 지원하는 키 저장소를 만듦.
- 하지만, 네트워크 장애는 피할 수 없는 일이기 때문에 분산 시스템은 Partition Tolerance를 고려해야함.
  - 즉, 실제로는 CA 시스템은 존재하지 않음.

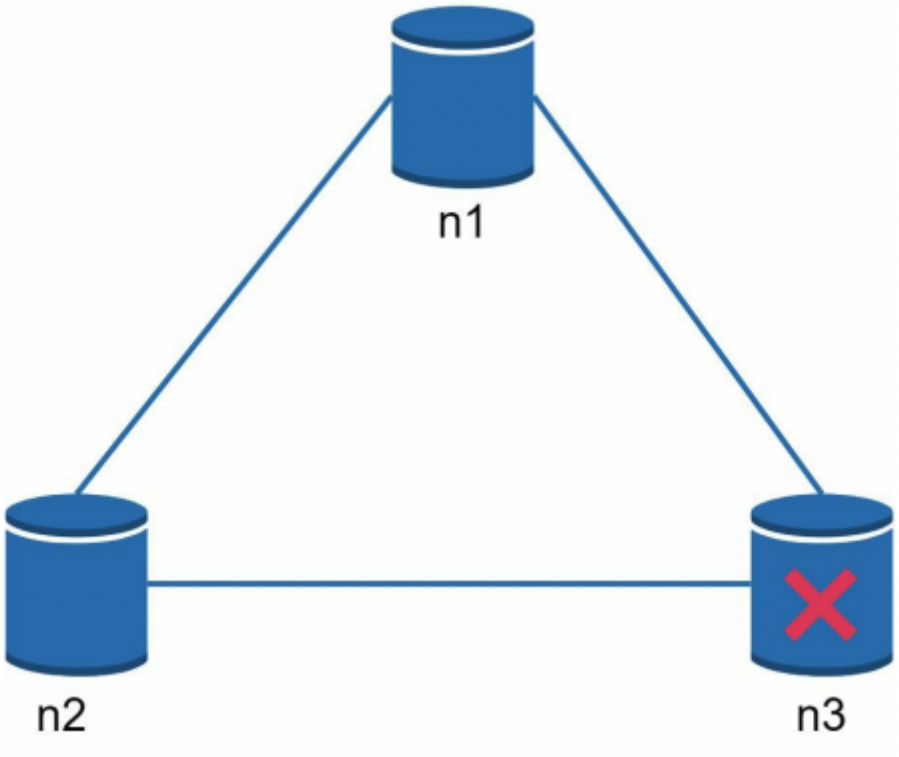
이상적 상태

- 네트워크가 파티션 되는 상황은 발생하지 않음.
- 즉, n1에 기록된 데이터는 자동적으로 n2, n3에 복제.
- 데이터 일관성 및 가용성도 만족됨.



실세계의 분산 시스템

- 분산시스템은 파티션 문제를 피할 수 없음.
- 파티션 문제가 발생하면, 일관성과 가용성 사이에 하나를 선택해야 함.
- 아래 그림에서, n1과 n2에 기록된 데이터는 n3에게 전달되지 않으며 반대의 경우도 전달되지 않음.



- 일관성을 선택한다면 (CP)
  - 데이터 불일치 문제를 피하기 위해, n1과 n2에 쓰기 연산을 중단시켜야 함. → 그렇지 않으면 가용성이 깨짐.
- 가용성을 선택한다면 (AP)
  - 읽기 연산을 계속 허용해야 함.
  - n1, n2는 계속 쓰기를 허용해야 하며, 파티션 문제가 해결되면 n3에 전송 될 것임.

시스템 컴포넌트

데이터 파티션

- 데이터를 여러 서버에 고르게 분산할 수 있는가
- 노드가 추가되거나 삭제 될 때, 데이터의 이동을 최소화 할 수 있는가

→ 이전 장에서 보았던 안정 해시가 이런 문제를 해결할 때 적합하다.

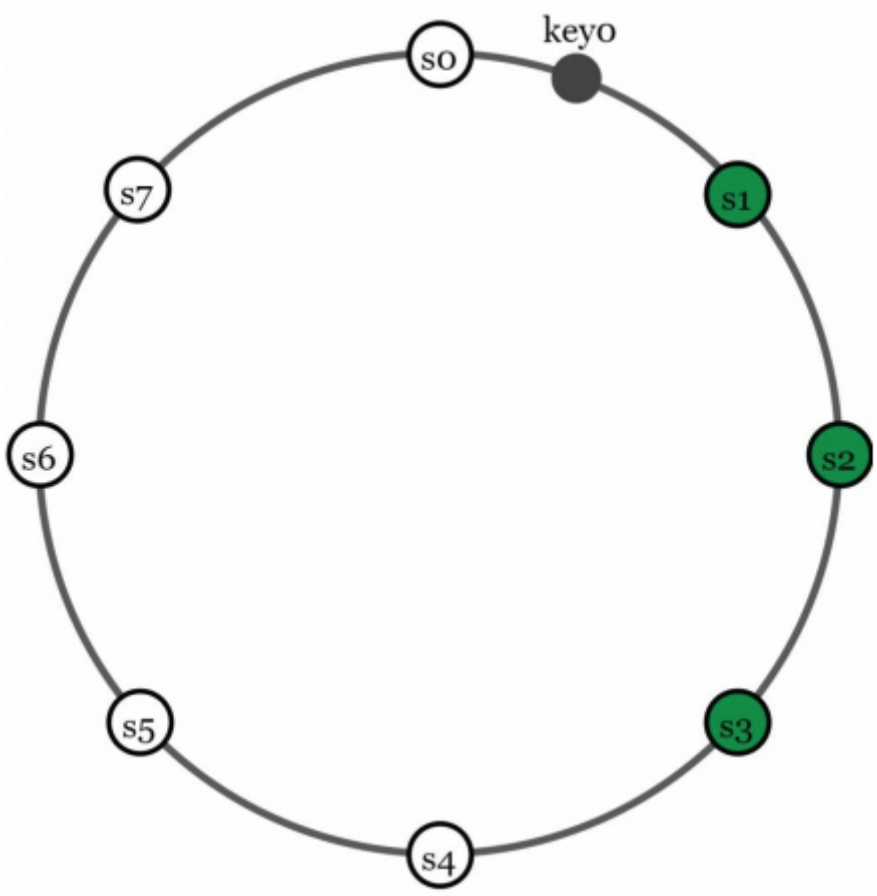
안정 해시는 데이터 파티션시 아래와 같은 장점이 있다.

- 규모 확장 자동화
  - 시스템 부하에 따라 서버가 자동으로 추가 혹은 삭제 될 수 있다.
- 다양성
  - 각 서버의 용량에 맞게 가상 노드의 수를 조정할 수 있다. 다시 말해, 고성능 서버는 더 많은 가상 노드를 갖도록 설정할 수 있다.

데이터 다중화

- 높은 가용성과 안정성을 확보하기 위해서는 데이터를 N개 서버에 비동기적으로 다중화(replication)할 필요가 있다.
  - N은 튜닝 가능한 값이며, 선정 방법은 아래와 같다.
  - 어떤 키를 해시 링위에 배치한 후, 그 지점으로부터 시계방향으로 순회하며 만나는 첫 N개 서버에 데이터 사본을 저장한다.

예시를 보면, N = 3일 경우 key 0 은 s1, s2, s3에 저장된다.

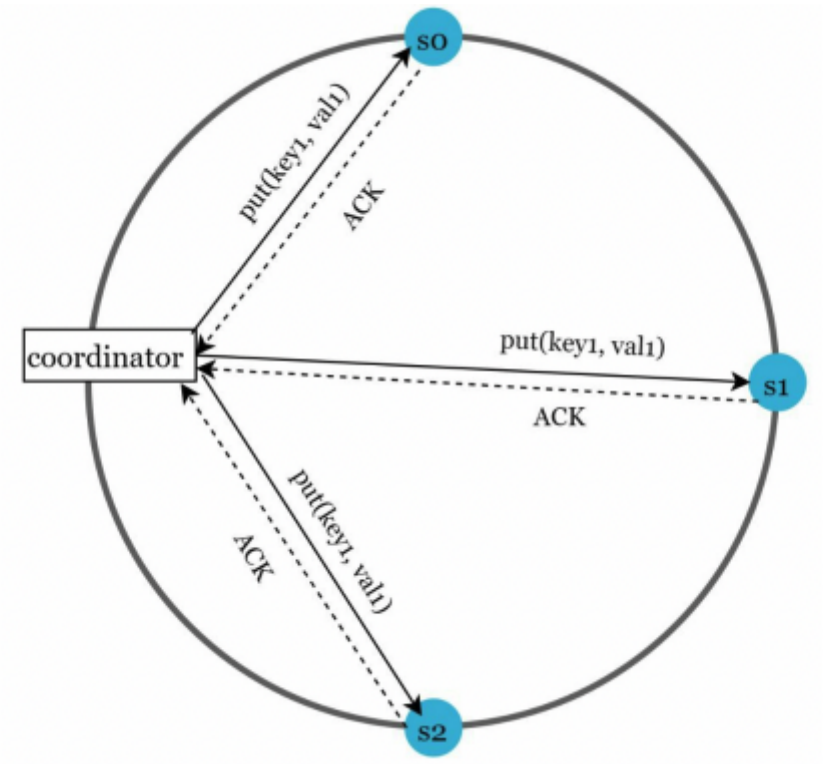


가상노드를 사용한다면, 같은 물리서버를 중복 선택하지 않도록 해야한다.

데이터의 사본들을 다른 센터의 서버로 다중화하고, 고속 네트워크 연결을하자.

데이터 일관성

- 여로 노드에 다중화된 데이터는 적절히 동기화 되어야한다.
- 정족수 합의 프로토콜을 사용하면, 읽기/쓰기 연산 모두에 일관성을 보장할 수 있다.



- N = 사본 개수
- W = 쓰기 연산에 대한 정족수. 쓰기 연산이 성공한 것으로 간주되려면 적어도 W개의 서버로부터 쓰기 연산이 성공했다는 응답을 받아야 한다.
- R = 읽기 연산에 대한 정족수. 읽기 연산이 성공한 것으로 간주되려면 적어도 R개의 서버로부터 응답을 받아야 한다.

W = 1 이라는 것은, 쓰기 연산이 성공했다고 최소 1대의 서버에서 중재자가 응답을 받아야한다는 것이다.

따라서, 중재자는 S1으로 부터 성공 응답을 받으면 S2, S3에게서 응답을 기다릴 필요가 없다.

→ 중재자는 클라이언트와 노드 사이에서 일종의 프록시 역할을 한다.

W, R, N 값을 찾는 것은 일종의 타협점을 찾아가는 과정이며, 간단하게 정리하면 아래와 같다.

- R = 1, W = N : 빠른 읽기 연산에 최적화된 시스템
- W = 1, R = N : 빠른 쓰기 연산에 최적화된 시스템
- W + R > N : 강한 일관성이 보장됨(보통 N=3, W = R = 2)
- W + R <= N : 강한 일관성이 보장되지 않음

강한 일관성이 무엇인지는 아래에서 알아볼 것이다.

일관성 모델

- 강한 일관성(strong consistency) : 모든 읽기 연산은 가장 최근에 갱신된 결과를 반환한다. 다시 말해서 클라이언트는 절대로 낡은 데이터를 보지 못한다.
- 약한 일관성(weak consistency) : 읽기 연산은 가장 최근에 갱신된 결과를 반환하지 못할 수 있다.
- 최종 일관성(eventual consistency) : 약한 일관성의 한 형태로, 갱신 결과가 결국에는 모든 사본에 반영(즉 ,동기화)되는 모델이다.

강한 일관성 모델은 새로운 쓰기 요청의 결과가 모든 노드에 반영될 때까지, 새로운 요청의 처리가 중단된다는 단점을 갖고 있어 고가용성 시스템에 적합하지 않다.

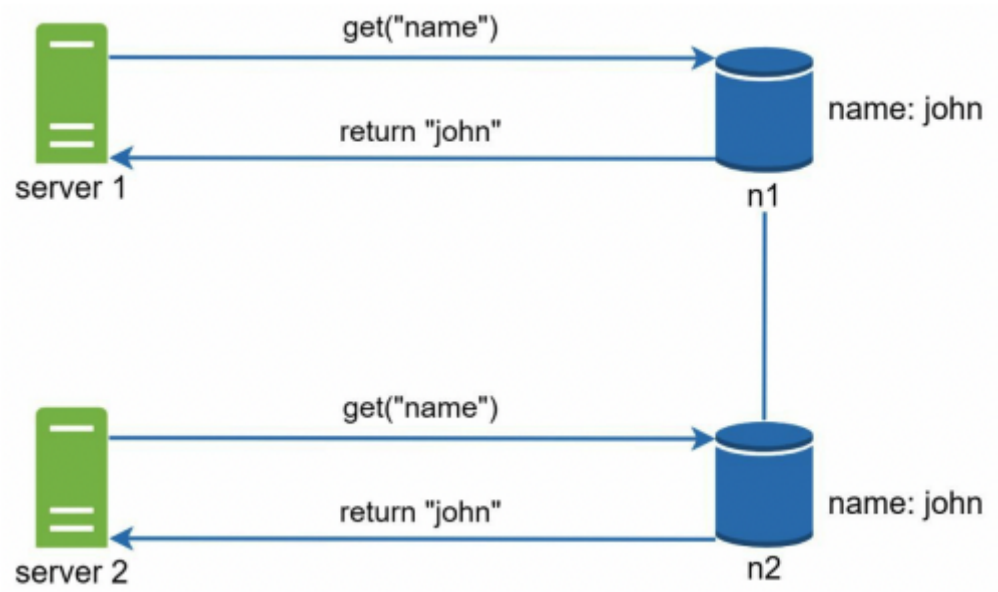
그렇기에 가장 많이 사용되는, 최종 일관성 모델을 사용하여 시스템을 설계할 것이다.

- 최종 일관성 모델 사용시 쓰기 연산이 병렬적으로 발생하면 시스템에 저장된 값의 일관성이 깨어질 수 있는데, 이 문제는 클라이언트가 해결해야 한다.
- 클라이언트측에서 데이터 버전 정보를 활용해 깨진 데이터를 읽지 않도록하는 기법을 아래에서 살펴보자.

비 일관성 해소 기법 : 데이터 버저닝

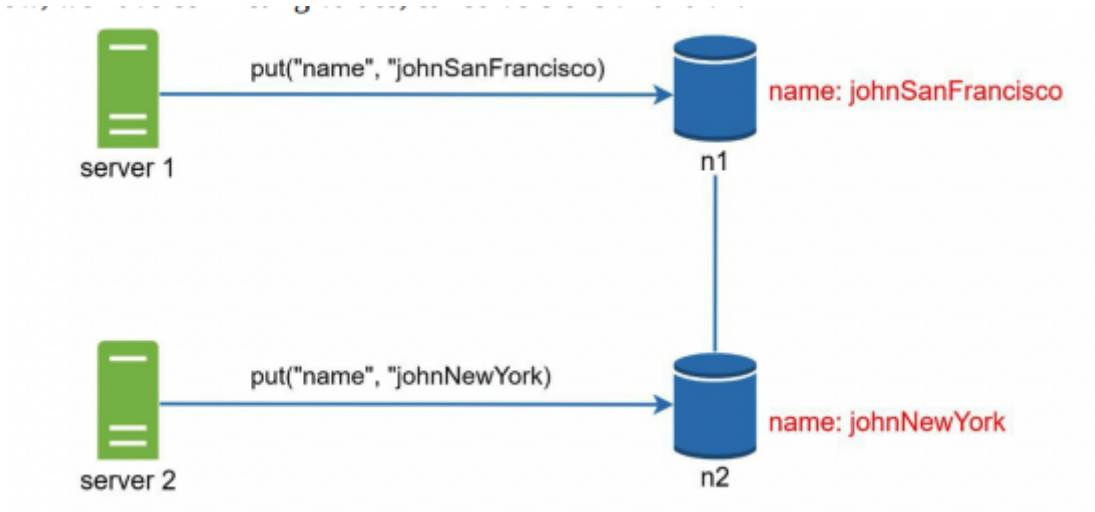
- 데이터 다중화를 이용하면 가용성은 높아지지만 일관성이 깨질 확률은 높아진다.
- 버저닝과 벡터 시계는 그 문제를 해소하기 위한 기술이며 알아보자.

버저닝은 데이터를 변경할 때마다 해당 데이터의 새로운 버전을 만드는 것을 의미한다. 따라서 각 버전의 데이터는 변경 불가능하다. 버저닝에 대해 자세히 알아보기 전, 어떠한 상황에서 일관성이 깨지는지부터 알아보자.



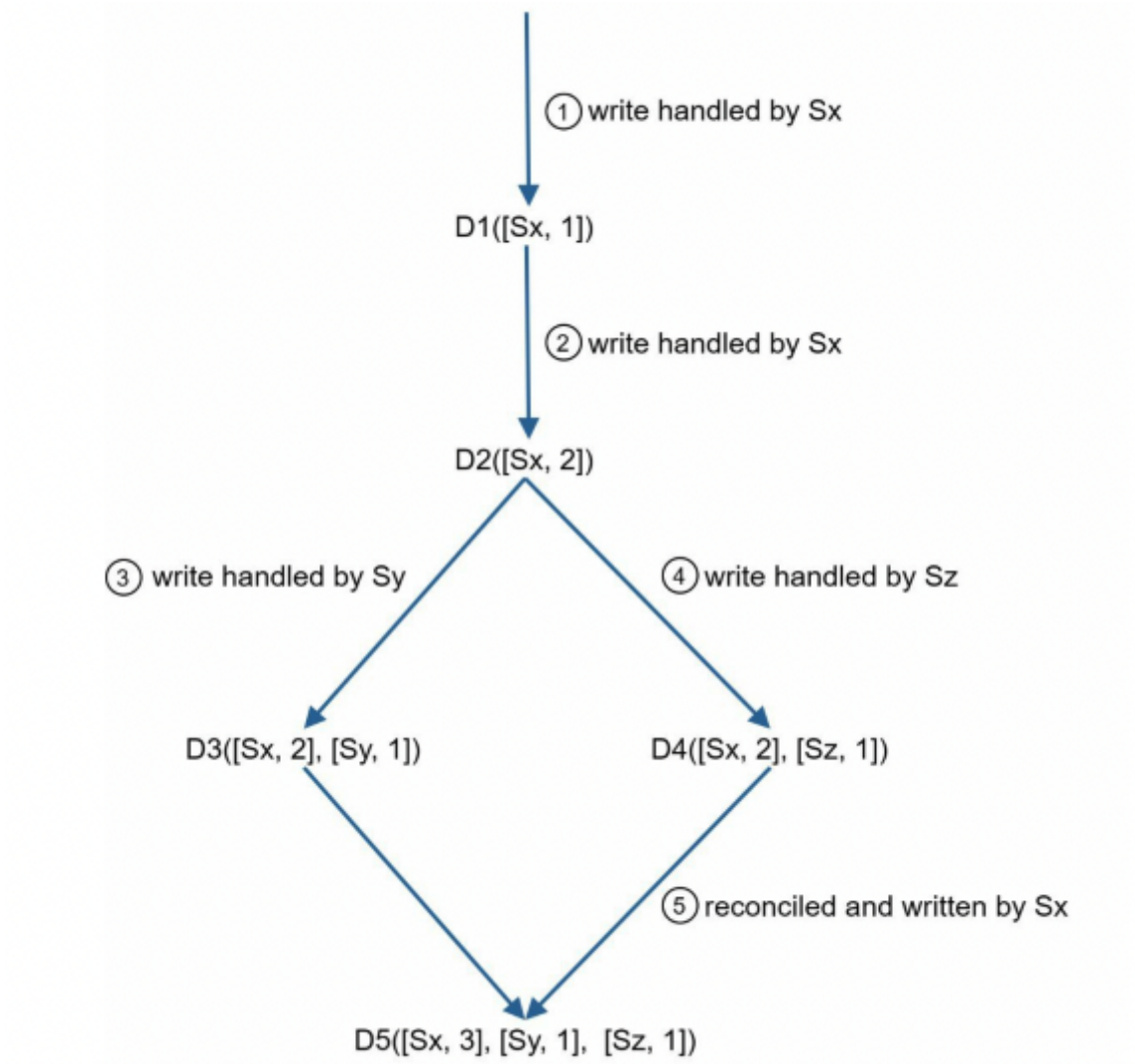
- 위와 같이, name이라는 key에 대해 jhon이라는 value를 가진 n1, n2 노드가 있다고 가정하자.

- 이때, 아래와 같이 server 1, server 2가 각각 name에 대해 서로 다른 value를 저장하면 conflict가 날 것이다.



- 이 두 개의 버전을 각각 v1, v2로 지정했을 때 충돌이 발생할 것이며 이것을 자동으로 해결할 수 있는 버저닝 시스템이 필요하다.
    - 이러한 상황을 해결할 수 있는 시스템이 벡터 시계이며, 동작 원리를 살펴보자.

- 벡터 시계
    - 벡터 시계는 [서버, 버전]의 순서쌍을 데이터에 매단 것이다.
    - 어떤 버전이 선행, 후행 버전인지 아니면 다른 버전과 충돌이 있는지 판별하는데 사용된다.
    - 벡터 시계는 D([S1, v1], [S2, v2], …, [Sn, vn]) 와 같이 표현한다고 가정하자.
        - D는 데이터
        - Sn 은 서버
        - vn은 버전
    - 만일 데이터 D를 서버 Si에 기록하면, 시스템은 아래 작업 가운데 하나를 수행하여야 한다.
        - [Si, vi]가 있으면 vi를 증가시킨다.
        - 그렇지 않으면 새 항목 [Si, 1]를 만든다.



위와 같은 플로우로 진행된다고 생각하면 된다.

1. 클라이언트가 데이터 D1을 시스템에 기록한다. 이 쓰기 연산을 처리한 서버는 Sx이다. 따라서 벡터 시계는 D1([Sx, 1])으로 변한다.
2. 다른 클라이언트가 데이터 D1을 읽고 D2로 업데이트한 다음 기록한다. D2는 D1에 대한 변경이므로 D1을 덮어 쓴다. 이때 쓰기 연산은 같은 서버 Sx가 처리한다고 가정하자. 벡터 시계는 D2([Sx, 2])로 바뀔 것이다.
3. 다른 클라이언트가 D2를 읽어 D3로 갱신한 다음 기록한다. 이 쓰기 연산은 Sy가 처리한다고 가정하자. 벡터 시계 상태는 D3([Sx, 2],[Sy 1])로 바뀐다.
4. 또 다른 클라이언트가 D2를 읽고 D4로 갱신한 다음 기록한다. 이 때 쓰기 연산은 서버 Sz가 처리한다고 가정하자. 벡터 시계는 D4([Sx, 2], [Sz, 1])일 것이다.
5. 어떤 클라이언트가 D3과 D4를 읽으면 데이터 간 충돌이 있다는 것을 알게 된다. D2를 Sy와 Sz가 각기 다른 값으로 바꾸었기 때문이다. 이 충돌은 클라이언트가 해소한 후에 서버에 기록한다. 이 쓰기 연산을 처리한 서버는 Sx였다고 하자. 벡터 시계는 D5([Sx, 3],[Sy, 1],[Sz, 1])로 바뀐다.
    - 충돌이 일어났다는 것을 어떻게 감지하는지는 잠시 후에 더 자세히 살펴볼 것이다.

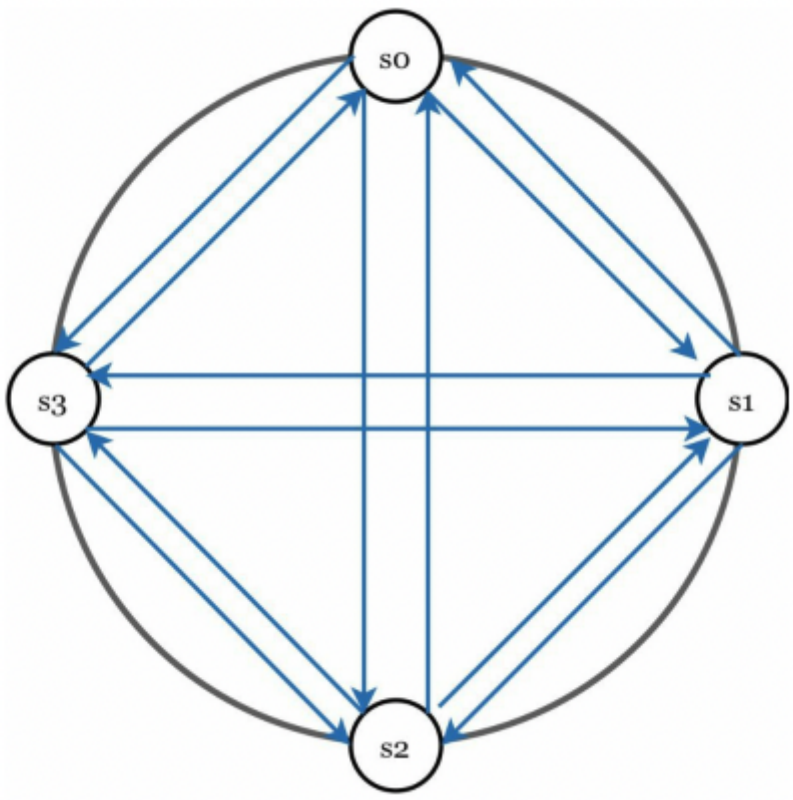
장애처리

장애가 발생했을 때, 장애를 감지하는 기법과 장애를 처리하는 장애 해소 기법이 있다.

우리는 장애 감지 기법을 먼저 살펴보고, 장애 해소 전략들을 살펴볼 것이다.

장애 감지

- 보통 분산 시스템에서 한 대의 서버가 죽었을 때, 바로 장애 처리를 하지 않는다.
- 일반적으로 두 대 이상의 서버가 똑같이 장애를 보고해야 실제로 장애가 발생했다 간주.
- 아래와 같이 모든 노드 사이에 멀티캐스팅 채널을 구축하는 것이 가장 쉬운 방법이지만, 비효율적.

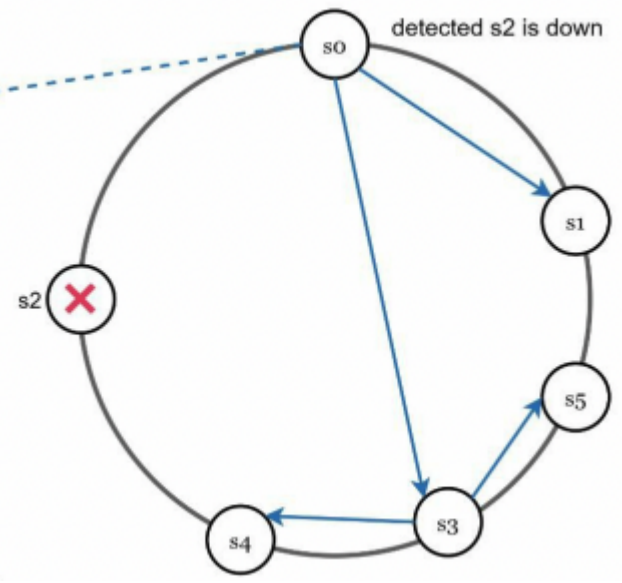


가십 프로토콜 같이 분산형 장애 감지 프로토콜을 사용해보자.

- 가십 프로토콜
    - 각 노드는 멤버십 목록을 유지한다. 멤버십 목록은 각 멤버 ID와 그 박동 카운터 쌍의 목록이다.
    - 각 노드는 주기적으로 자신의 박동 카운터를 증가시킨다.
    - 각 노드는 무작위로 선정된 노드들에게 주기적으로 자기 박동 카운터 목록을 보낸다.
    - 박동 카운터 목록을 받은 노드는 멤버십 목록을 최신 값으로 갱신한다.
    - 어떤 멤버의 박동 카운터 값이 지정된 시간 동안 갱신되지 않으면 해당 멤버는 장애 상태인 것으로 간주한다.

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



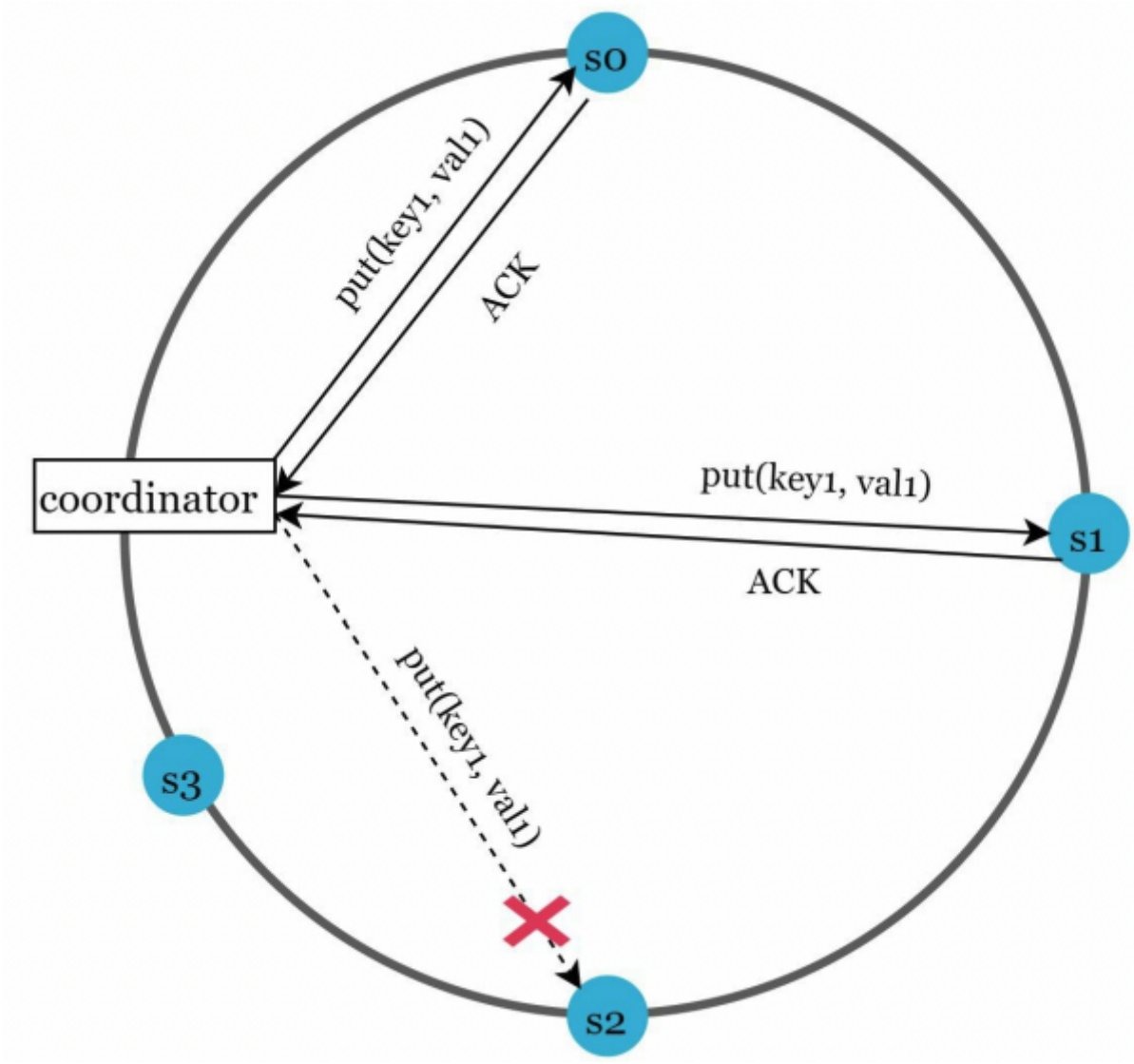
- 노드 s0은 그림 좌측의 테이블과 같은 멤버십 목록을 가진 상태다.
- 노드 s0은 노드 s2(멤버 ID=2)의 박동 카운터가 오랫동안 증가되지 않았다는 걸 발견한다.
- 노드 s0은 노드 s2를 포함하는 박동 카운터 목록을 무작위로 선택된 다른 노드에게 전달한다.
- 노드 s2의 박동 카운터가 오랫동안 증가되지 않았음을 발견한 모든 노드는 해당 노드를 장애 노드로 표시한다.

일시적 장애 처리

가십 프로토콜로 장애를 감지한 시스템은 가용성을 보장하기 위해 필요한 조치를 해야 한다.

엄격한 정족수(strict quorum) 접근법을 쓴다면, 앞서 데이터 일관성에서 설명한 대로, 읽기와 쓰기 연산을 금지해야 할 것이다. 느슨한 정족수(sloppy quorum) 접근법은 이 조건을 완화하여 가용성을 높인다.

- 느슨한 정족수 접근법
  - 정족수 요구사항을 강제하는 대신, 쓰기 연산을 수행할 W개의 건강한 서버와 읽기 연산을 수행할 R개의 건강한 서버를 해시 링에서 고른다. 이때 장애 상태인 서버는 무시한다.
  - 네트워크나 서버 문제로 장애 상태인 서버로 가는 요청은 다른 서버가 잠시 맡아 처리한다.
  - 그동안 발생한 변경사항은 해당 서버가 복구되었을 때 일괄 반영하여 데이터 일관성을 보존한다.
  - 이를 위해 임시로 쓰기 연산을 처리한 서버에는 그에 관한 단서를 남겨둔다.
  - 이런 장애 처리 방안을 단서 후 임시 위탁(hinted handoff) 기법이라고 부른다.



위 그림을 보면, s2에 대한 읽기 및 쓰기 연산은 일시적으로 s3가 처리하다 복구되면 s3는 갱신된 데이터를 s2로 인계할 것이다.

영구적 장애 처리

단서 후 위탁 처리는 임시 장애 처리를 위해 사용했다. 만약 서버가 영구적으로 장애가 발생했다면, 다른 방식을 취해야한다.

이러한 상황을 처리하기 위해 반-엔트로피(anti-entropy) 프로토콜을 구현하여 사본들을 동기화 한다.

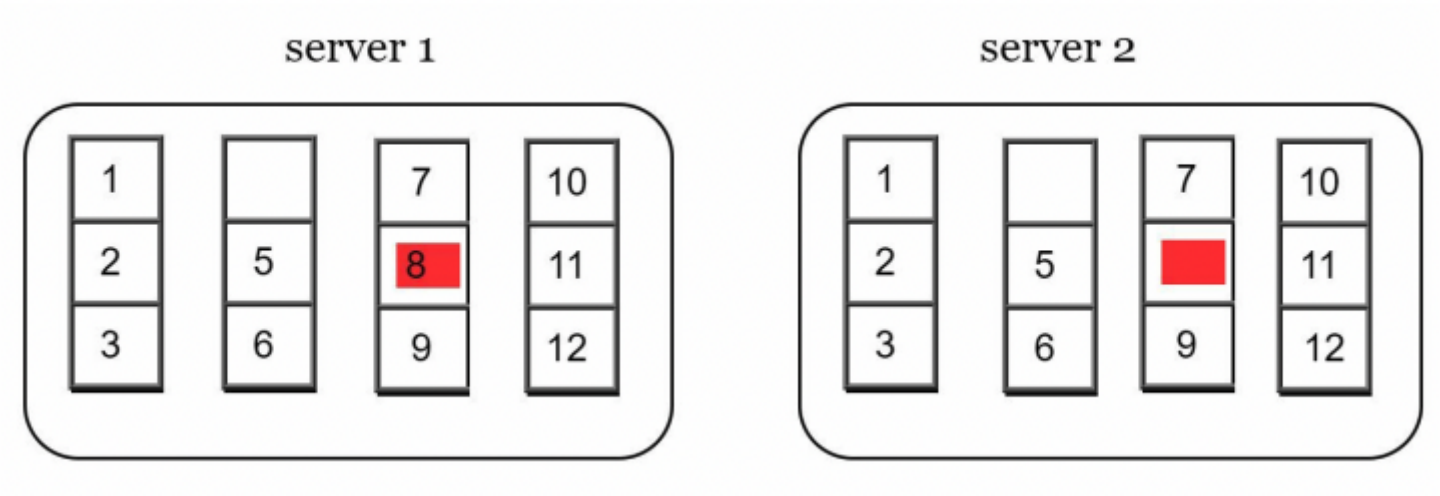
반 엔트로피 프로토콜은 사본들을 비교하여 최신 버전으로 갱신하는 과정을 포함한다.

사본 간의 일관성이 망가진 상태를 탐지하고 전송 데이터의 양을 줄이기 위해서는 머클트리를 사용할 것이다.

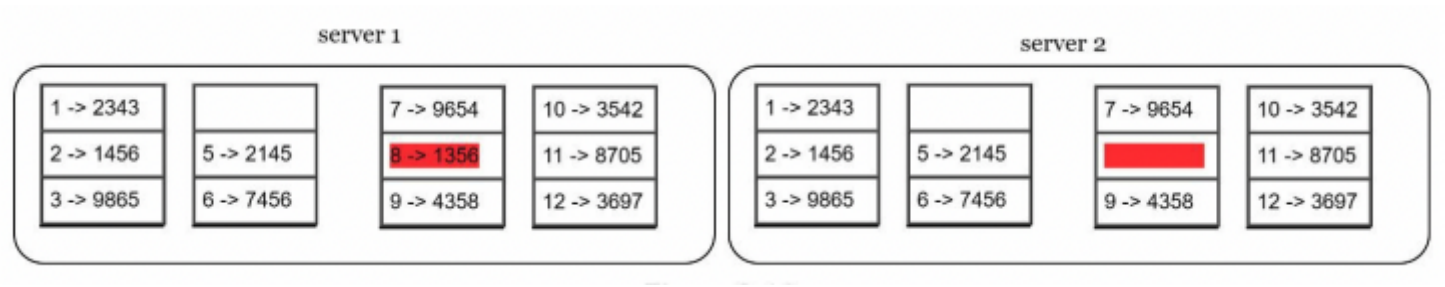
- 머클 트리
  - 해시 트리라고도 불리는 머클 트리는 각 노드에 그 자식 노드들에 보관된 값의 해시(자식 노드가 종단 노드인 경우), 또는 자식 노드들의 레이블로부터 계산된 해시 값을 레이블로 붙여두는 트리다.
  - 해시 트리를 사용하면 대규모 자료 구조의 내용을 효과적이면서도 보안상 안전한 방법으로 검증할 수 있다.

예시로 살펴보자. (일관성이 망가진 데이터가 위치한 상자는 다른 색으로 표시되어있다.)

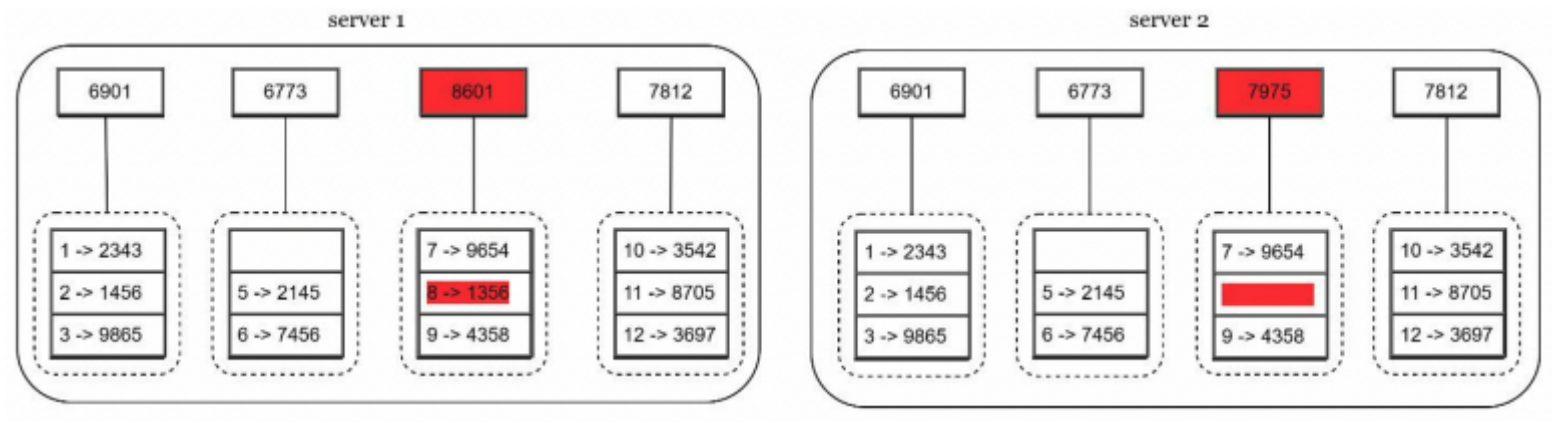
1단계 : 아래 그림과 같이 버킷으로 나눈다.



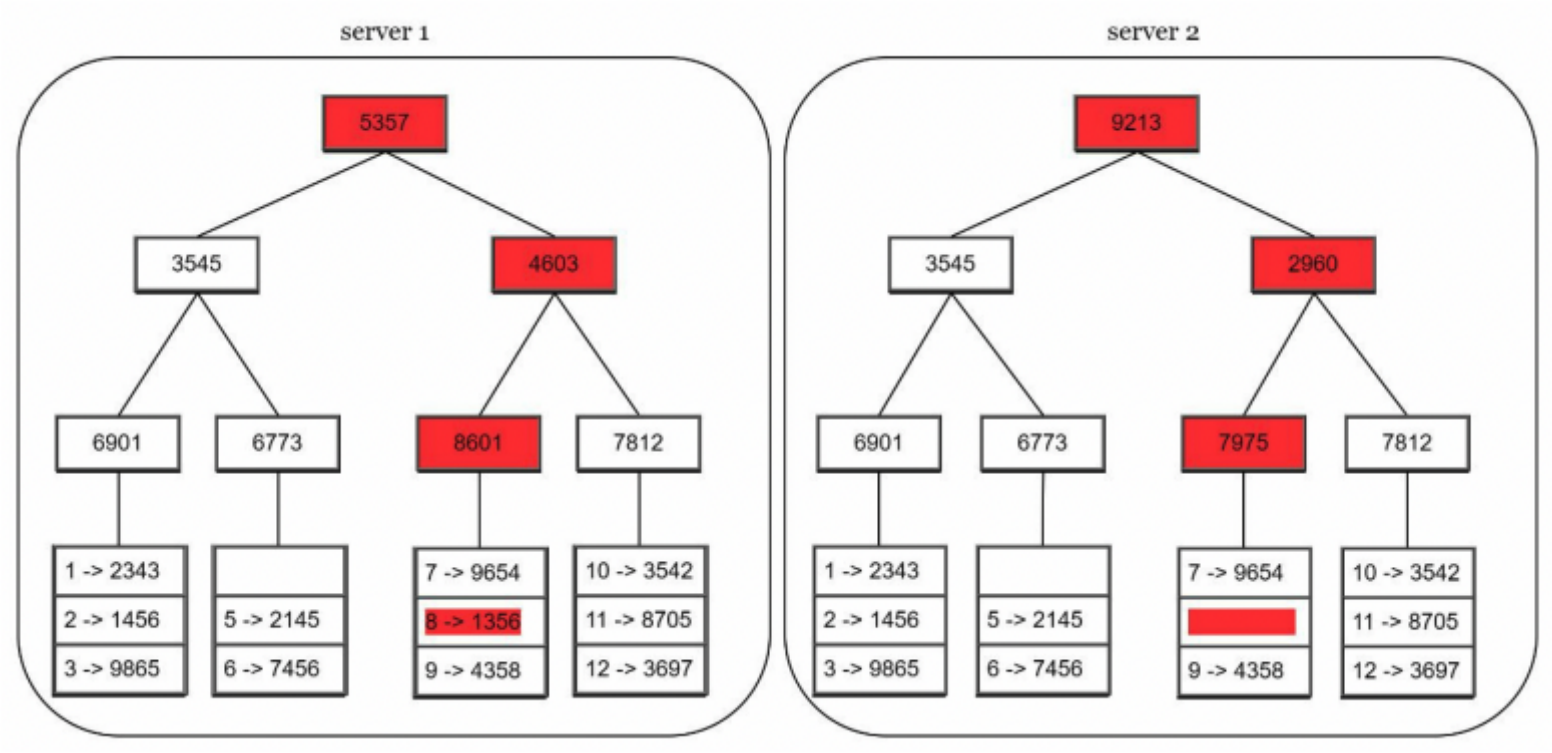
2단계 : 버킷에 포함된 각각의 키에 균등 분포 해시를 적용하여 해시 값을 계산한다.



3단계 : 버킷별로 해시값을 계산한 후, 해당 해시 값을 레이블로 갖는 노드를 만든다.



4단계 : 자식 노드의 레이블로부터 새로운 해시 값을 계산하여, 이진트리를 상향식으로 만들어간다.



이 두 머클 트리의 비교는 루트 노드의 해시값을 비교하는 것으로 시작한다.

- 루트 노드의 해시 값이 일치한다면 두 서버는 같은 데이터를 갖는 것이다.
- 만약 루트 노드의 값이 다르다면, 왼쪽 노드의 해시 값을 비교한다.
- 왼쪽 노드의 해시 값이 같다면, 오른쪽 노드의 해시 값을 비교한다.
- 이렇게 점점 아래로 비교해가면서, 서로 다른 데이터를 가지는 버킷을 찾아낼 수 있고 이를 통해 동기화하면 된다.

머클 트리를 사용하면 동기화해야 하는 데이터의 양은 실제로 존재하는 차이의 크기에 비례할 뿐, 두 서버에 보관된 데이터의 총량과는 무관해진다.

- 하지만 실제로 쓰이는 시스템의 경우 버킷 하나의 크기가 꽤 크다는 것을 알아두어야 한다.
- 예를 들면 10억개의 키를 백만개의 버킷으로 관리한다면, 하나의 버킷은 1,000개 키를 관리하게 된다.

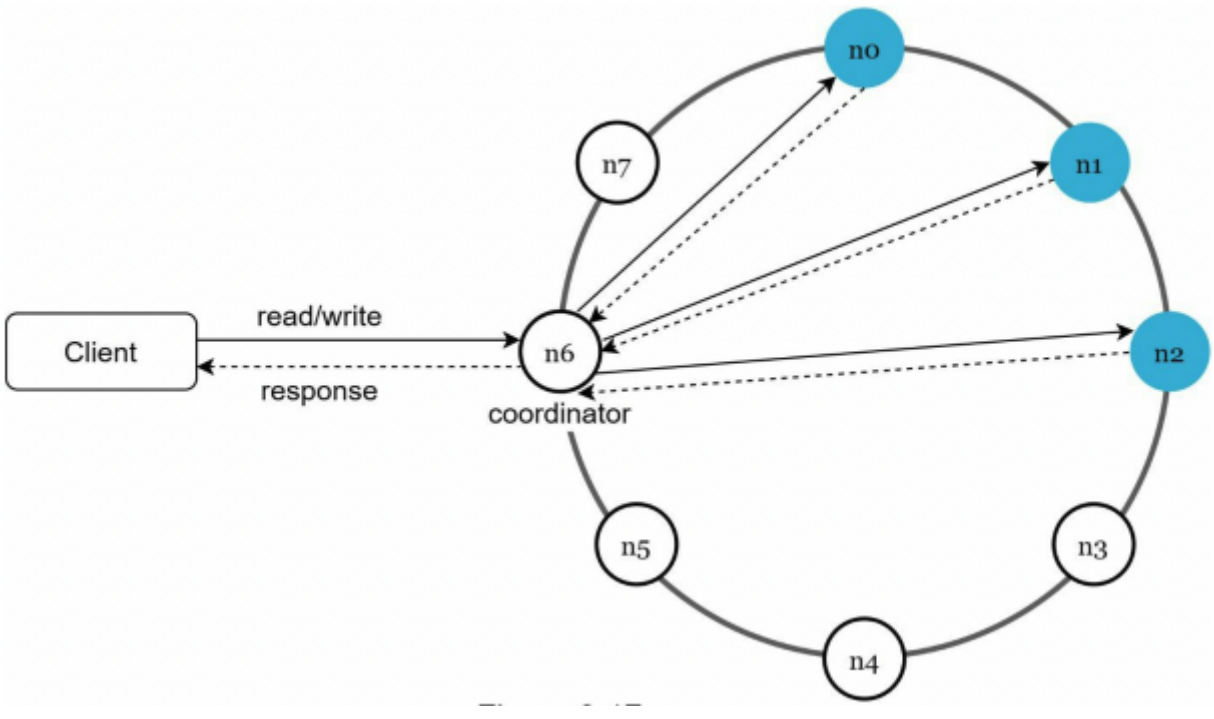
데이터 센터 장애 처리

한 데이터 센터에만 데이터를 저장하면, 자연 재해등의 발생시 문제가 발생한다.

여러 데이터 센터에 다중화하는 방안으로 데이터 센터 장애처리를 하자.

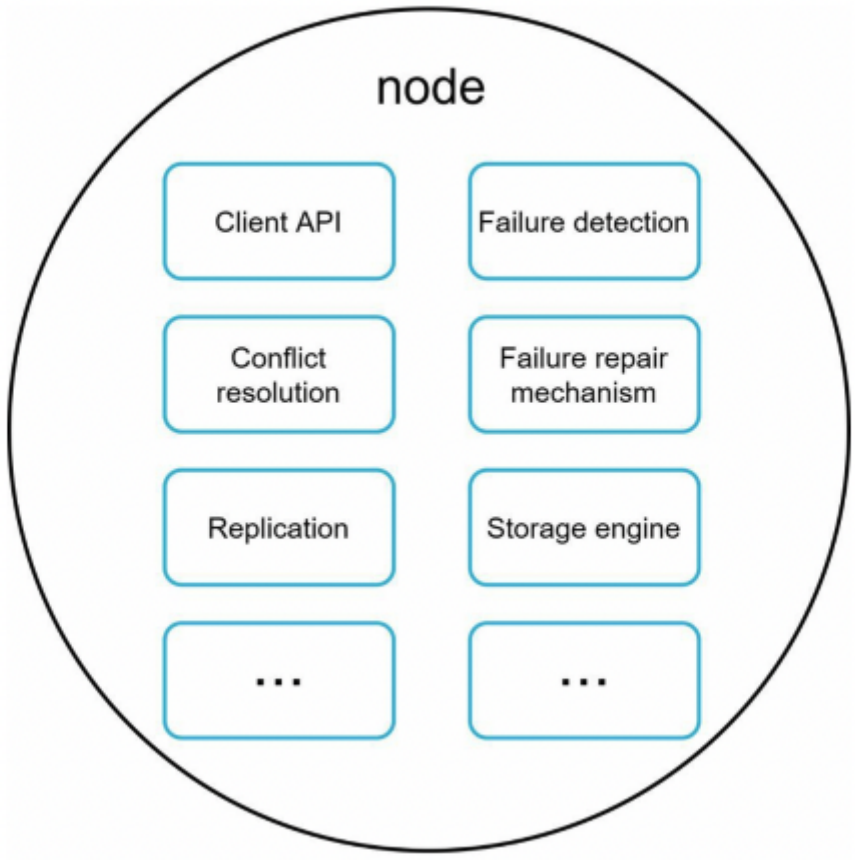
시스템 아키텍쳐 다이어그램

다양한 기술들을 고려해보았으니, 이제 아키텍쳐 다이어그램을 그려보자.



- 클라이언트는 키-값 저장소가 제공하는 두 가지 단순한 API, 즉 get 및 put와 통신한다.
- 중재자는 클라이언트에게 키-값 저장소에 대한 proxy 역할을 하는 노드이다.
- 노드는 안정 해시의 해시링 위에 분포한다.
- 노드를 자동으로 추가 또는 삭제할 수 있도록 시스템은 완전히 분산된다.
- 데이터는 여러 노드에 다중화된다.
- 모든 노드가 같은 책임을 지므로, SPOF는 존재하지 않는다.

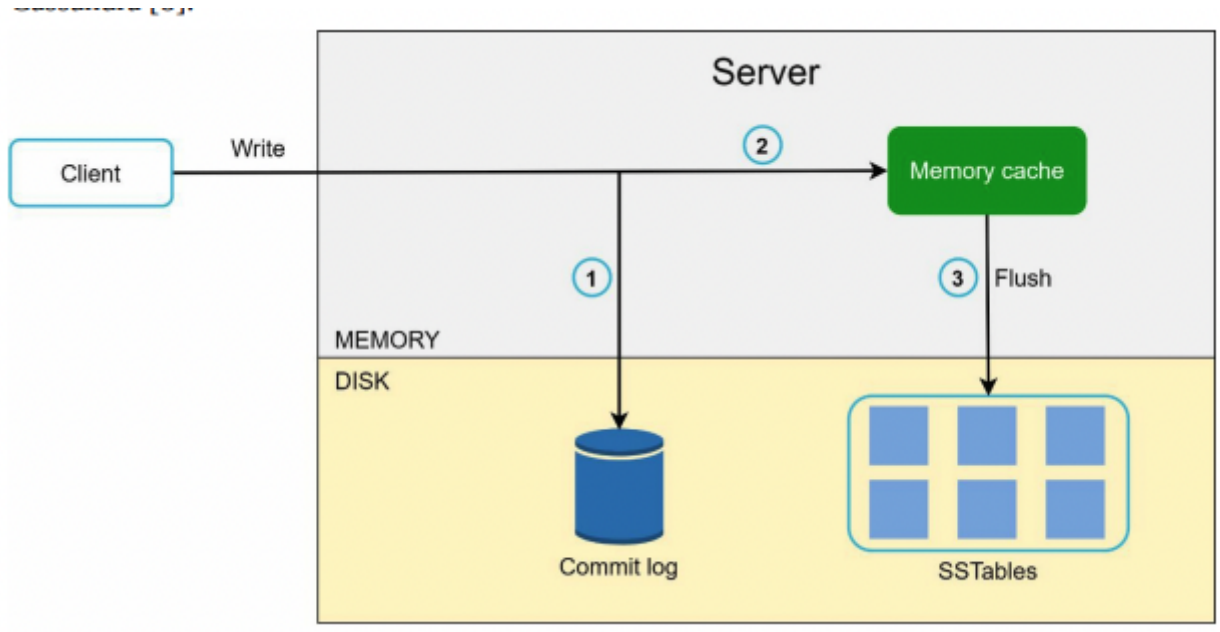
완전히 분산된 설계를 채택하였으므로, 모든 노드는 아래 그림에 제시된 기능을 전부 지원해야한다.



쓰기 경로

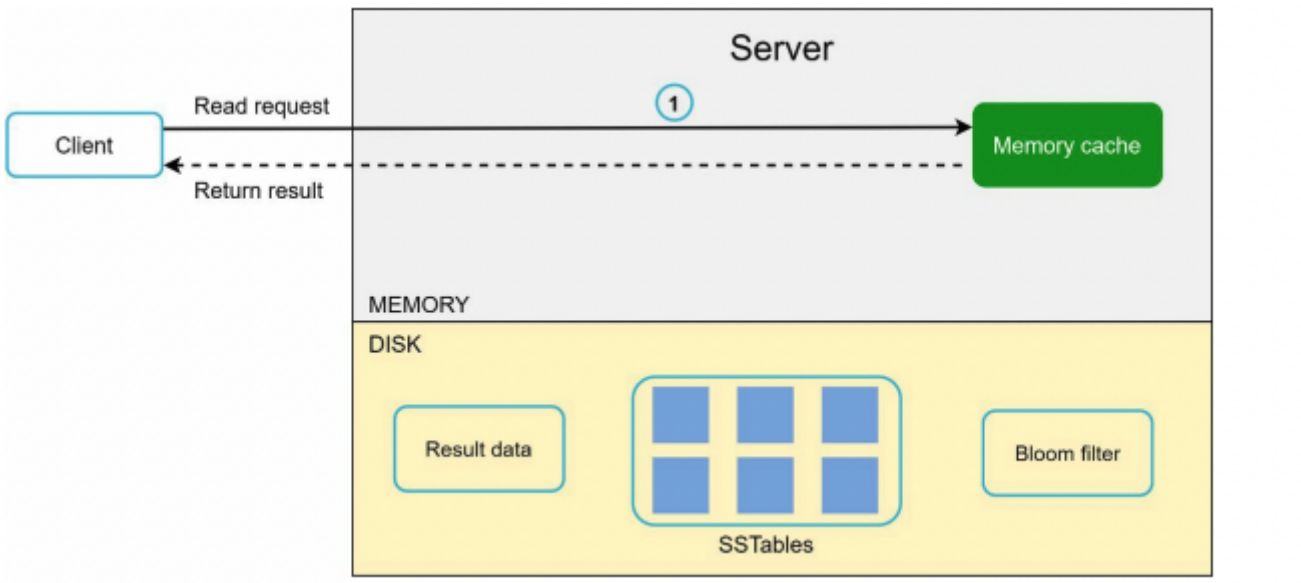
아래 그림은, 쓰기 요청이 특정 노드에 전달되면 무슨 일이 벌어지는지를 보여준다.

1. 쓰기 요청이 커밋 로그 파일에 기록된다.
2. 데이터가 메모리 캐시에 기록된다.
3. 메모리 캐시가 가득하거나 임계치에 도달하면 데이터는 디스크에 있는 SSTable에 기록된다.
    - SSTable 은 Sorted-String Table의 약어로 key-value의 쌍을 리스트 형태로 관리하는 테이블이다.

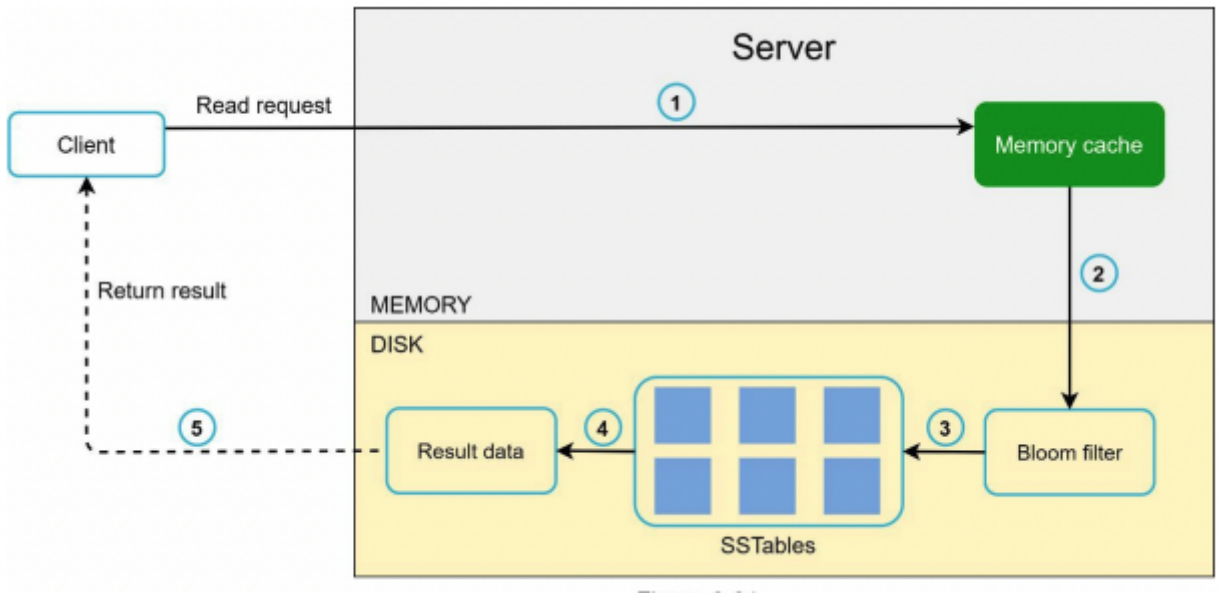


읽기 경로

- 읽기 요청을 받은 노드는 데이터가 메모리 캐시에 있는지부터 살핀다. 있는 경로라면, 데이터를 클라이언트에 반환하며 메모리에 없는 경우 디스크에서 가져온다.



- 이때, 어느 SSTable에 존재하는지 효율적으로 찾기 위해 블룸 필터를 사용하며 아래의 그림과 같이 이루어진다.



1. 데이터가 메모리 있는지 검사한다. 없으면 2로 간다.
2. 데이터가 메모리에 없으므로 블룸 필터를 검사한다.
3. 블룸 필터를 통해 어떤 SSTable에 키가 보관되어 있는지 알아낸다.
4. SSTable에서 데이터를 가져온다.
5. 해당 데이터를 클라이언트에게 반환한다.

요약

여러 개념과 기술을 다루었다. 간단하게 정리해보자.

- 대규모 데이터 저장
  - 안정 해시를 사용해 서버들에 부하 분산
- 읽기 연산에 대한 높은 가용성 보장
  - 데이터를 여러 데이터센터에 다중화
- 쓰기 연산에 대한 높은 가용성 보장
  - 버저닝 및 벡터 시계를 활용
- 데이터 파티션
  - 안정 해시
- 점진적 규모 확장성
  - 안정 해시
- 다양성
  - 안정 해시

- 조절 가능한 데이터 일관성
  - 정족수 합의
- 일시적 장애 처리
  - 느슨한 정족수 프로토콜과 단서 후 임시 위탁
- 영구적 장애 처리
  - 머클 트리
- 데이터 센터 장애 대응
  - 여러 데이터 센터에 걸친 다중화