5️⃣

231112 15장. 구글 드라이브 설계

▼ 문제 이해 및 설계 범위 확정

기증적 요구사항

- 파일 추가
  - 가장 쉬운 방법은 구글 드라이브 안으로 drag-and-drop
- 파일 다운로드
- 여러 단말에 파일 동기화
  - 한 단말에서 파일 추가 시 다른 단말에도 자동으로 동기화
- 파일 갱신 이력 조회
- 파일 공유
- 파일이 편집되거나 삭제되거나 새롭게 공유되었을 때 알림 표시

비기능적 요구사항

- 안정성
  - 저장소 시스템에서 안정성은 매우 중요
  - 데이터 손실 발생하면 안됨
- 빠른 동기화 속도
- 네트워크 대역폭
  - 너무 많이 사용하면 사용자 불편 초래
- 규모 확장성
  - 아주 많은 양의 트래픽도 처리 가능해야 함
- 높은 가용성

- 일부 서버에 장애가 발생하거나, 느려지거나, 네트워크 일부가 끊겨도 시스템은 계속 사용 가능해야 함

개략적 추정치

- 가입 사용자 오천만명, 천만 명의 DAU
- 모든 사용자에게 10GB의 무료 저장공간 할당
- 매일 각 사용자가 평균 2개의 파일을 업로드한다고 가정하고 각 파일의 평균 크기는 약 500KB
- 읽기:쓰기 비율은 1:1
- 필요한 저장공간 총량 = 오천만 사용자 * 10GB = 500페타바이트(Petabyte)
- 업로드 API QPS = 천만 사용자 * 2회 업로드 / 24시간 / 3600초 = 약 240
- 최대 QPS = OPS * 2 = 480

▼ 개략적 설계안 제시 및 동의 구하기

▼ 서버

1. 파일을 올리고 다운로드 하는 과정을 처리할 웹 서버
2. 사용자 데이터, 로그인 정보, 파일 정보 등의 메타데이터를 보관할 데이터베이스
3. 파일을 저장할 저장소 시스템
    - 파일 저장을 위해 1TB의 공간을 사용할 것

▼ API

파일 업로드 API

- 단순 업로드
    - 파일 크기가 작을 때 사용
- 이어 올리기
    - 파일 사이즈가 크고 네트워크 문제로 업로드가 중단될 가능성이 높다고 생각되면 사용함
    - 절차
        1. 이어 올리기 URL을 받기 위한 최초 요청 전송
        2. 데이터를 업로드하고 업로드 상태 모니터링
        3. 업로드에 장애가 발생하면 장애 발생시점부터 업로드를 재시작

파일 다운로드 API

```
{
    "path": "/recipes/soup/best_soup.txt"
}

// path - 다운로드할 파일의 경로
```

파일 갱신 히스토리 제공 API

```
{
    "path": "/recipes/soup/best_soup.txt",
    "limit": 20
}

// path - 갱신 히스토리를 가져올 파일의 경로
// limit - 히스토리 길이의 최대치
```

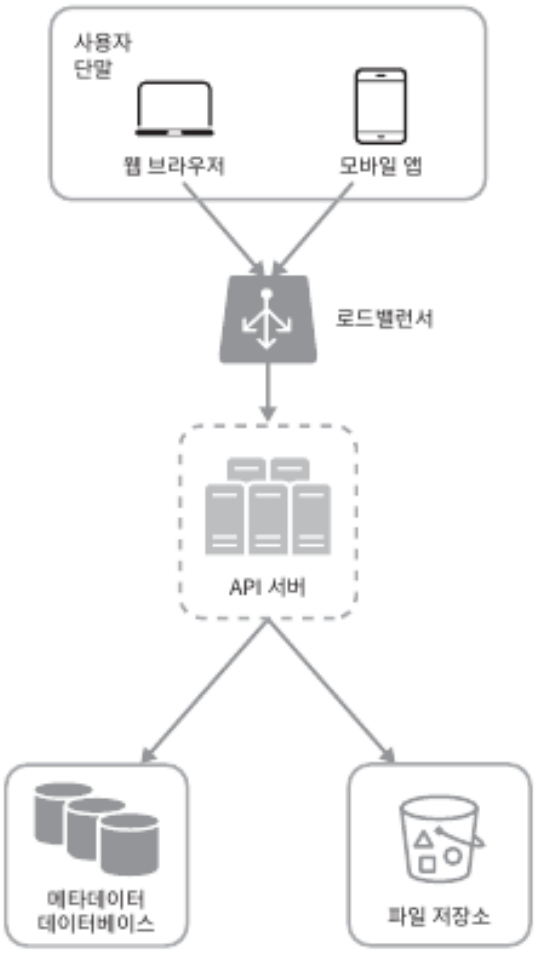
- 모든 API는 사용자 인증을 필요로 함
- HTTPS 프로토콜을 사용해야 함
- SSL을 지원하는 프로토콜을 이용하는 이유
    - 클라이언트와 백엔드 서버가 주고받는 데이터를 보호하기 위해

▼ 한 대 서버의 제약 극복

업로드되는 파일이 많아지다 보면 파일 시스템이 가득 차게 됨

- 샤딩
- 아마존 S3
    - 넷플릭스, 에어비엔비 같은 시장 주도 기업들이 많이 상요함

    - 업계 최고 수준의 규모 확장성, 가용성, 보안, 성능을 제공하는 객체 저장소 서비스
    - 다중화 지원
        - 같은 지역 안에서 다중화 가능
        - 여러 지역에 걸쳐 다중화 가능
            - 데이터 손실을 막고 가용성을 최대한 보장할 수 있음
- 로드밸런서
    - 네트워크 트래픽을 분산
    - 웹 서버에 장애가 발생하면 자동으로 해당 서버를 우회
- 웹 서버
    - 로드밸런스를 추가하면 더 많은 웹 서버를 손쉽게 추가 가능
    - 트래픽이 폭증해도 쉽게 대응 가능
- 메타데이터 데이터베이스
    - 데이터베이스를 파일 저장 서버에서 분리하여 SPOF를 회피함
    - 다중화 및 샤딩 정책을 적용하여 가용성과 규모 확정성 요구사항에 대응
- 파일 저장소
    - S3를 파일 저장소로 사용하고 가용성과 데이터 무손실을 보장하기 위해 두 개 이상의 지역에 데이터를 다중화함



위 내용을 모두 적용한 수정 설계안

▼ 동기화 충돌

두 명 이상의 사용자가 같은 파일이나 폴더를 동시에 업데이트 하려고 할 때

먼저 처리되는 변경 → 성공

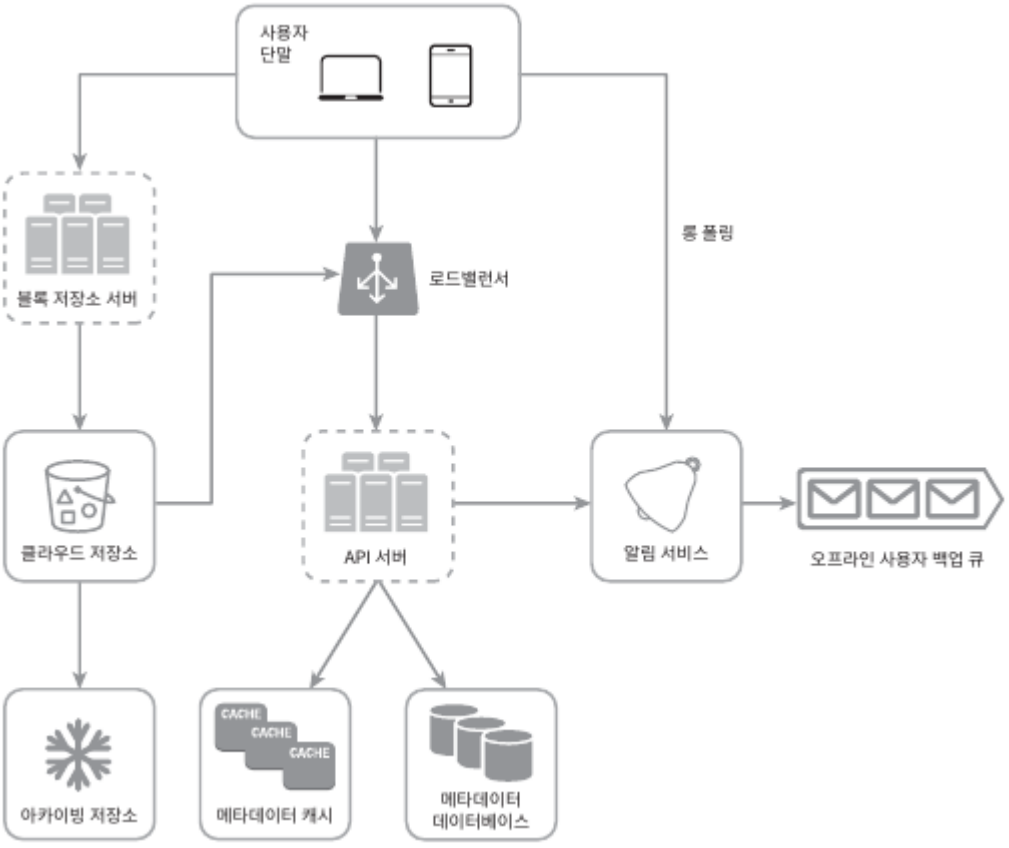
나중에 처리되는 변경 → 충돌 발생

충돌이 발생한 파일에 대한 처리 방법

1. 두 파일을 하나로 합치기
2. 둘 중 하나를 다른 파일로 대체하기

⇒ 시스템에 맞게 결정하기

▼ 개략적 설계안



개략적 설계안

- 사용자 단말
  - 사용자가 이용하는 웹브라우저나 모바일 앱 등의 클라이언트
- 블록 저장소 서버(= 블록 수준 저장소)
  - 파일 블록을 클라우드 저장소에 업로드하는 서버
  - 클라우드 환경에서 데이터 파일을 저장하는 기술
  - 파일을 여러 개의 블록으로 나누어 저장하며 각 블록에는 고유한 해시값이 할당됨
  - 해시값은 메타데이터 데이터베이스에 저장
  - 각 블록은 독립적인 객체로 취급되면 클라우드 저장소(S3)에 보관됨
- 클라우드 저장소
  - 파일은 블록 단위로 나눠져 클라우드 저장소에 보관
- 아카이빙 저장소

    - 오랫동안 사용되지 않은 비활성 데이터를 저장하기 위한 컴퓨터 시스템
- 로드밸런서
- API 서버
    - 파일 업로드 외에 거의 모든 것을 담당하는 서버
    - 사용자 인증, 사용자 프로파일 관리, 파일 메타데이터 갱신 등에 사용
- 메타데이터 데이터베이스
    - 사용자, 파일, 블록, 버전 등의 메타데이터 정보를 관리
    - 실제 파일은 클라우드에 보관
- 메타데이터 캐시
    - 자주 쓰이는 메타데이터는 캐싱
- 알림 서비스
    - 특정 이벤트가 발생했음을 클라이언트에게 알리는ㄴ데 쓰이는 sub/pub 프로토콜 기반 시스템
- 오프라인 사용자 백업 큐
    - 클라이언트가 접속 중이 아니라서 파일의 최신 상태를 확인할 수 없을 때는 해당 정보를 이 큐에 두어 나중에 클라이언트가 접속했을 때 동기화될 수 있도록 함

▼ 상세 설계

▼ 블록 저장소 서버

정기적으로 갱신되는 큰 파일들을 업데이트가 일어날 때마다 전체 파일을 서버로 보내면 네트워크 대역폭을 많이 잡아먹게 됨

최적화 방법

- 델타 동기화
    - 파일이 수정되면 전체 파일 대신 수정이 일어난 블록만 동기화
- 압축
    - 블록 단위로 압축해 두면 데이터 크기를 많이 줄일 수 있음
    - 압축 알고리즘은 파일 유형에 따라 지정

블록 저장소 서버는 파일 업로드에 관계된 힘든 일을 처리하는 컴포넌트

- 블록 저장소 서버는 클라이언트가 보낸 파일을 블록 단위로 나눠야 함
- 각 블록에 압축 알고리즘을 적용하고 암호화까지 해야 함
- 전체 파일을 저장소 시스템으로 보내는 대신 수정된 블록만 전송해야 함

동작법

1. 주어진 파일을 작은 블록들로 분할
2. 각 블록을 압축
3. 암호화
4. 클라우드 저장소 전송

▼ 높은 일관성 요구사항

강한 일관성 모델을 기본으로 지원해야 함

⇒ 같은 파일이 단말이나 사용자에 따라 다르게 보이는 것은 허용할 수 없음

⇒ 메타데이터 캐시와 데이터베이스 계층에도 같은 원칙이 적용되어야 함

메모리 캐시

- 보통 최종 일관성 모델을 지원
- 캐시에 보관된 사본과 데이터베이스에 있는 원본이 일치
- 데이터베이스에 보관된 원본에 변경이 발생하면 캐시에 있는 사본을 무효화함

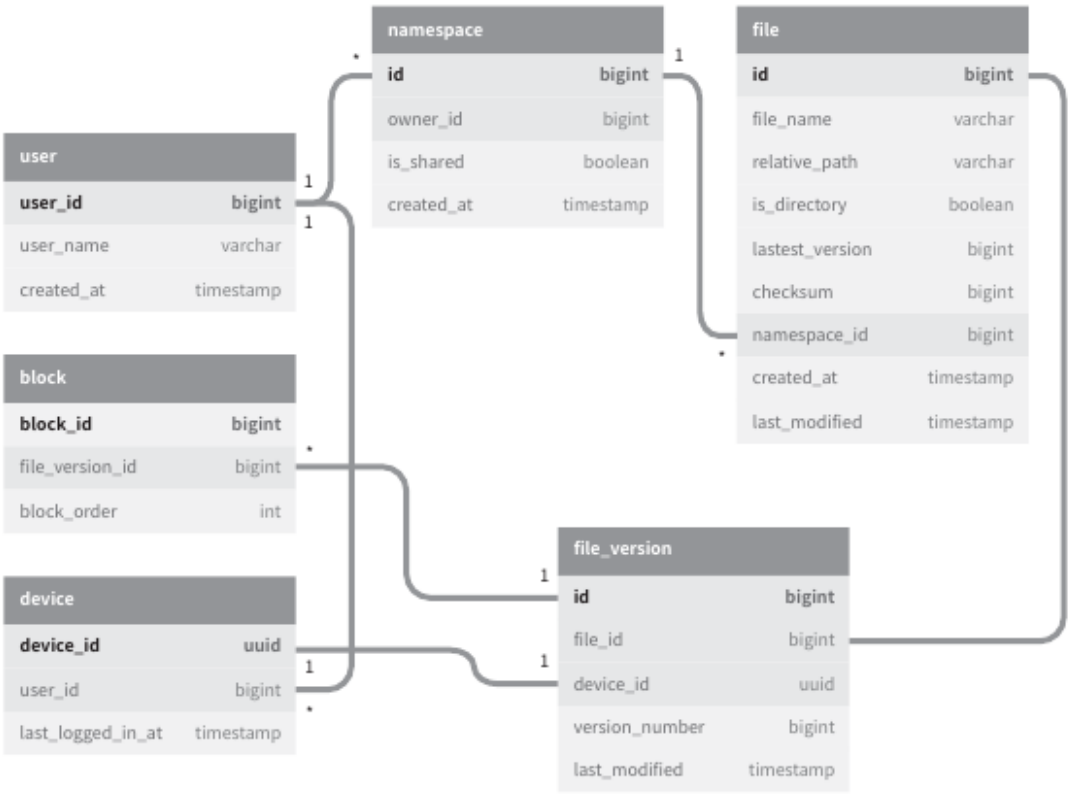
관계형 데이터베이스

- ACID를 보장하기 때문에 강한 일관성 보장하기 쉬움

NoSql

- ACID를 지원하지 않기 때문에 동기화 로직 안에 프로그램해 넣어야 함

▼ 메타데이터 데이터베이스



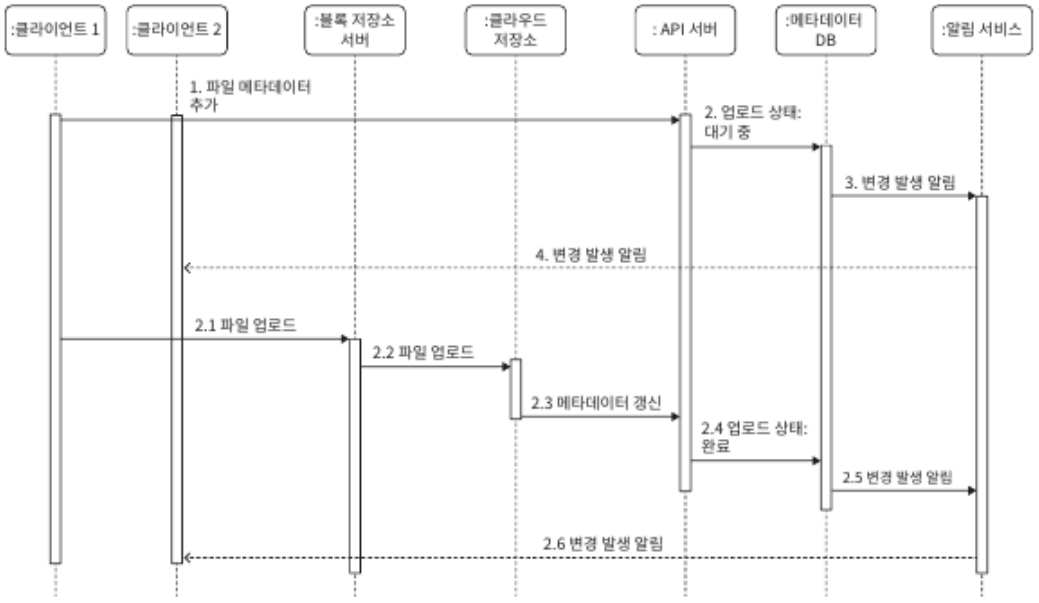
스키마 설계안

- user
  - 이름, 이메일, 프로파일 사진 등 사용자에 관계된 기본적 정보들이 보관
- device
  - 단말 정보가 보관됨
  - push_id → 모바일 푸시 알림을 보내고 받기 위한 것
  - 한 사용자가 여러 대의 단말을 가질 수 있음에 유의
- namespace
  - 사용자의 루트 디렉터리 정보가 보관됨
- file
  - 파일의 최신 정보가 보관됨
- file_version
  - 파일의 갱신 이력이 보관되는 테이블
  - 읽기 전용
  - 갱신 이력이 훼손되는 것을 막기 위해

- block
  - 파일 블록에 대한 정보를 보관하는 테이블
  - 특정 버전의 파일은 파일 블록을 올바른 순서로 조합하기만 하면 복원 가능

▼ 업로드 절차

두 개 요청이 병렬적으로 전송된 상황



- 파일 메타데이터 추가
  1. 클라이언트 1이 새 파일의 메타데이터를 추가하기 위한 요청 전송
  2. 새 파일의 메타데이터를 데이터베이스에 저장하고 업로드 상태를 대기중으로 변경
  3. 새 파일이 추가되었음을 알림 서비스에 통지
  4. 알림 서비스는 관련된 클라이언트(클라이언트 2)에게 파일이 업로드되고 있음을 알림
- 파일을 클라우드 저장소에 업로드
  1. 클라이언트 1이 파일을 블록 저장소 서버에 업로드
  2. 블록 저장소 서버는 파일을 블록 단위로 쪼갠 다음 압축하고 암호화 한 다음에 클라우드 저장소에 전송
  3. 업로드가 끝나면 클라우드 스토리지는 완료 콜백을 호출
  4. 콜백 호출은 API 서버로 전송됨

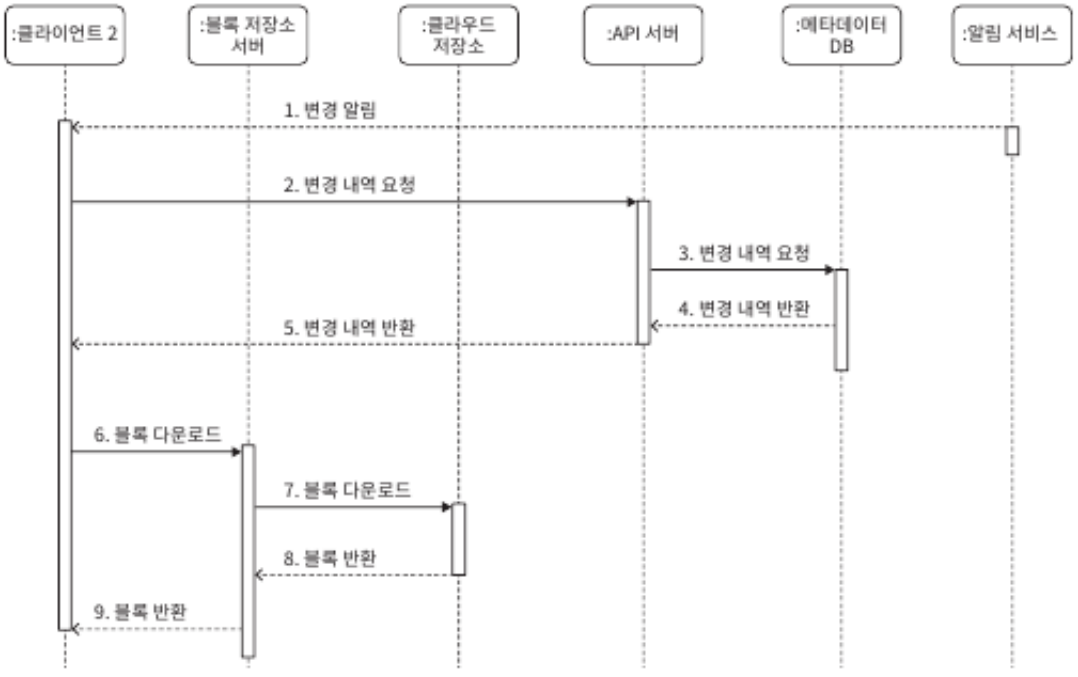
5. 메타데이터 DB에 기록된 해당 파일의 상태를 완료로 변경
6. 알림 서비스에 파일 업로드가 끝났음을 통지
7. 알림 서비스는 관련된 클라이언트에게 파일 업로드가 끝났음을 알림

▼ 다운로드 절차

파일이 새로 추가되거나 편집되면 자동으로 시작됨

다른 클라이언트가 파일을 편집하거나 추가했다는 사실을 감지하는 방법

- 클라이언트 A가 접속 중이고 다른 클라이언트가 파일을 변경하면 알림 서비스가 클라이언트 A에게 변경이 발생했으니 새 버전을 끌어가야 한다고 알림
- 클라이언트 A가 네트워크에 연결된 상태가 아닐 경우에는 데이터는 캐시에 보관될 것이고 해당 클라이언트의 상태가 접속 중으로 바뀌면 그때 해당 클라이언트는 새 버전을 가져갈 것



1. 알림 서비스가 클라이언트 2에게 누군가 파일을 변경했음을 알림
2. 알림을 확인한 클라이언트 2는 새로운 메타데이터를 요청
3. API 서버는 메타데이터 데이터베이스에게 새 메타데이터를 요청
4. API 서버에게 새 메타데이터가 반환됨

5. 클라이언트 2에게 새 메타데이터가 반환됨
6. 클라이언트 2는 새 메타데이터를 받는 즉시 블록 다운로드 요청 전송
7. 블록 저장소 서버는 클라우드 저장소에서 블록 다운로드
8. 클라우드 저장소는 블록 서버에 요청된 블록 반환
9. 블록 저장소 서버는 클라이언트에게 요청된 블록 반환
10. 클라이언트 2는 전송된 블록을 사용하여 파일 재구성

▼ 알림 서비스

알림 서비스의 목적

- 파일의 일관성을 유지하기 위해, 클라이언트는 로콜에서 파일이 수정되었음을 감지하는 순간 다른 클라이언트에 그 사실을 알려서 충돌 가능성을 줄여야 함

이벤트 데이터를 클라이언트들로 보내는 서비스

- 롱 폴링
  - 드롭박스가 이 방식을 채택
  - 이번 설계안에서 채택
    - 양방향 통신이 필요하지 않음
    - 알림을 많이 보내지 않음
    - 각 클라이언트는 알림 서버와 롱 폴링용 연경을 유지하다가 특정 파일에 대한 변경을 감지하면 해당 연결을 끊고, 클라이언트는 메타데이터 서버와 연결해 파일의 최신 내역을 다운로드 해야 함
    - 다운로드 작업이 끝났거나 연결 타임아웃 시간에 도달한 경우에는 즉시 새 요청을 보내어 롱 폴링 연결을 복원하고 유지해야 함
- 웹소켓
  - 클라이어느와 서버 사이에 지속적인 통신 채널을 제공
  - 양방향 통신 가능

▼ 저장소 공간 절약

파일 갱신 이력을 보존하고 안정성을 보장하기 위해서는 파일의 여러 버전을 여러 데이터센터에 보관할 필요가 있는데 모든 버전을 자주 백업하게 되면 저장용량이 너무 빨리 소진될 가능성이 있음

- 중복 제거
  - 중복된 파일 블록을 계정 차원에서 제거하는 방법
  - 해시값 비교
- 지능적 백업 전략 도입
  - 한도 설정
    - 보관해야 하는 파일 버전 개수에 상한을 두는 것
    - 상한에 도달하면 제일 오래된 버전은 버림
  - 중요한 버전만 보관
    - 불필요한 버전과 사본이 만들어지는 것을 피하려면 그 가운데 중요한 것만 골라내야 함
- 자주 쓰이지 않는 데이터는 아카이빙 저장소로 이관

▼ 장애 처리

- 로드밸런서 장애
  - 로드 밸런서끼리는 보통 박동 신호를 주기적으로 보내서 상태를 보니터링함
  - 일정 시간동안 박동 신호에 응답하지 않은 로드밸런서는 장애가 발생한 것으로 간주함
- 블록 저장소 서버 장애
  - 블록 저장소 서버에 장애가 발행하였다면 다른 서버가 미완료 상태 또는 대기 상태인 작업을 이어받아야 함
- 클라우드 저장소 장애
  - S3 버킷은 여러 지역에 다중화할 수 있으므로, 한 지역에서 장애가 발생하였다면 다른 지역에서 파일을 가져오면 됨
- API 서버 장애
  - API 서버들은 무상태 서버
  - 로드밸런서는 API 서버에 장애가 발생하면 트래픽을 해당 서버로 보내지 ㅇ 낳음으로써 장애 서버를 격리할 것
- 메타데이터 캐시 장애
  - 메타데이터 캐시 서버도 다중화
  - 한 노드에 장애가 생겨도 다른 노드에서 데이터를 가져올 수 있음

    - 장애가 발생한 서버는 새 서버로 교체하면 됨
- 메타데이터 데이터베이스 장애
    - 주 데이터베이스 서버 장애
        - 부 데이터베이스 서버 가운데 하나를 주 데이터베이스 서버로 바꾸고, 부 데이터베이스 서버를 새로 하나 추가함
    - 부 데이터베이스 서버 장애
        - 다른 부 데이터베이스 서버가 읽기 연산을 처리하도록 하고 그동안 서버는 새 것으로 교체함
- 알림 서비스 장애
    - 접속 중인 모든 사용자는 알림 서버와 롱 폴링 연결을 하나씩 유지함
    - 알림 서비스는 많은 사용자와의 연결을 유지하고 관리해야 함
    - 한 대 서버에 장애가 발생하면 백만 명 이상의 사용자가 롱 폴링 연결을 다시 만들어야 함
    - 주의할 것은 한 대 서버로 백만 개 이상의 접속을 유지하는 것은 가능하지만, 동시에 백만 개 접속을 시작하는 것은 불가능

  ⇒ 롱 폴링 연결을 복구하는 것은 상대적으로 느릴 수 있음
- 오프라인 사용자 백업 큐 장애
    - 다중화해 두어야 함
    - 큐에 장애가 발생하면 구독 중인 클라이언트들은 백업 큐로 구독 관계를 재설정해야 할 것

▼ 마무리

이번 설계안의 큰 부분들

- 파일의 메타데이터를 관리
- 파일 동기화를 처리
- 알림 서비스
    - 두 부분과 병존하는 또 하나의 중요 컴포넌트
    - 롱 폴링을 사용하여 클라이언트로 하여금 파일의 상태를 최신으로 유지할 수 있도록 함

ex) 블록 저장소 서버를 거치지 않고 파일을 클라우드 저장소에 직접 업로드

- 파일 전송을 클라우드 저장소로 직접 하면 되기 때문에 업로드 시간이 빨라질 수 있음
- 분할, 압축, 암호화 로직을 클라이언트에 두어야 하기 때문에 플랫폼별로 따로 구현해야 함
- 클라이언트가 해킹 당할 가능성이 있기 때문에 암호화 로직을 클라이언트 안에 두는 것은 적절치 않은 선택일 수도 있음

접속 상태를 관리하는 로직을 별도 서비스로 옮기기

- 관련 로직을 알림 서비스에서 분리해 내면 다른 서비스에서도 쉽게 활용할 수 있음

▼ 토론

충돌이 발생한 파일에 대한 처리 방법이 두가지 있는데 어떤 경우에 각각 써야 하는지

1. 두 파일을 하나로 합치기
2. 둘 중 하나를 다른 파일로 대체하기

구글 스프레드시트

1. **두 파일을 하나로 합치는 경우:**
    - **상황:** 여러 사용자가 동시에 문서를 편집할 때 충돌이 발생한 경우.
    - **예시:** Google Docs에서 동시에 편집 중인 두 사용자의 변경 사항을 문서 하나로 통합하는 경우. 각 변경 사항을 비교하고 합병하여 최종 문서를 생성.
2. **둘 중 하나를 다른 파일로 대체하는 경우:**
    - **상황:** 두 사용자가 동일한 부분을 수정하거나 경쟁적인 변경이 있을 때.
    - **예시:** Dropbox에서 동일한 파일을 동시에 수정한 경우, 최신 버전을 선택하고 다른 사용자의 변경 사항을 덮어씌우는 방식으로 충돌을 해결.