5️⃣

15장 구글 드라이브 설계

1단계 문제 이해 및 설계 범위 확정

구글 드라이브는 클라우드 저장소다. AWS의 S3 같은 것도 클라우드 저장소이며, 네이버의 마이 박스 등 우리의 삶에 클라우드 저장소는 깊게 들어와 사용되고 있다.

기본적으로 클라우드 저장소를 설계하는 것은, 큰 프로젝트다.

문서, 사진, 비디오 등 대부분의 데이터를 보관할 수 있어야 하며 스마트폰, 태블릿, PC 등 다양한 플랫폼을 지원해야 하는 등 만능에 가까운 저장소의 기능을 지원할 수 있어야 한다.

그러나, 면접장에서 만능에 가까운 무언가를 구현하기를 아무도 기대하지 않으리라 자신한다. 당신이 할 일은 당신에게 기대하는 설계 범위 내에서 최선의 저장소를 설계하는 것이다. 책에서는 구글 드라이브로 표현 했지만 사실은 클라우드 저장소 설계와 같다.

책에서는 다음과 같은 설계를 원한다.

- 가장 중요한 기능은 파일 업로드/다운로드/파일 동기화/알림이다.
- 모바일 앱과 웹 둘 다 지원해야 한다.
- 파일 암호화가 필요하다.
- 파일 크기는 최대 10GB이다.

- 사용자는 DAU(일간) 천만명이다.

위 요구사항에 맞춰 기능 설계에 집중하면,

일단 파일 추가, 가장 쉬운 방법은 데이터를 drag-and-drop 방식으로 끌어당겨 올려놓는 것일 거다.

한 단말에 파일을 업로드하면, 자동으로 다른 단말에서도 동기화된 파일을 다운로드하고 볼 수 있어야 한다.

또한, 파일이 새롭게 업로드되거나 삭제 수정 됐을 때 알람도 필요하다.

위 같은 기능적 요구사항외에도 저장소 설계에는 비-기능적 요구사항도 중요하다.

- 안정성
- 빠른 동기화 속도
- 네트워크 대역폭 ⇒ 이 제품이 불필요한 대역폭 소모를 한다면 사용자 경험이 떨어질 것이다.
- 규모 확장성
- 높은 가용성

개략적 추정치는 다음과 같다.

- 가입자는 5천만명, 하루 천만 명의 DAU 발생
- 모든 사용자에게 10GB 무료 저장공간 할당
- 매일 각 사용자가 2개의 파일을 업로드 한다고 치면, 각 파일의 평균 크기는 500kb
- 읽기 : 쓰기 비율은 1:1
- 필요 저장 공간은 5천만 사용자 * 10GB = 500 petabyte
- 업로드 API QPS = 1천만 사용자 * 2회 업로드 / 24시간 / 3600 초 = 약 240
- 최대 QPS = QPS * 2 = 480

2단계 개략적 설계안 제시 및 동의 구하기

15장에선 컴포넌트부터 제시했던 것과 달리, 단일서버에서 출발한다.

일단 다음과 같은 구성으로 시작한다.

- 업로드와 다운로드를 책임 질 웹 서버
- 사용자 정보 등을 보관할 데이터베이스
- 파일을 저장할 저장소 시스템

웹 서버의 API들은 어떤 기능들을 제공해야 할까?

API

기본적으로 API는 3가지의 기능을 제공해야 한다.

1. 파일 업로드

   이 시스템은 기본적으로 두 종류의 업로드를 지원한다.

   - 단순 업로드 : 파일 크기가 작을 때 사용한다.
   - 이어 올리기 : 파일 사이즈가 크거나 네트워크로 인해 업로드가 중단될 경우를 고려한다.

   https://api.example.com/files/upload?uploadType=resumable

   기본 인자는 uploadType과 data가 있다.

   이어 올리기는 다음 세 단계 절차로 이루어져 있다.

   - 이어 올리기 URL을 받기 위한 최초 요청 전송
   - 데이터를 업로드하고 업로드 상태 모니터링
   - 업로드에 장애가 발생하면 장애 발생지점부터 업로드를 재시작

2. 파일 다운로드

   https://api.example.com/files/download

3. 파일 갱신 히스토리

   https://api.example.com/files/list_revisions

https 써라...

한 대 서버의 제약 극복

파일이 많아지다보면 결국 스토리지는 꽉 찰 것이다. 이렇게 되면 더는 업로드할 수 없다.

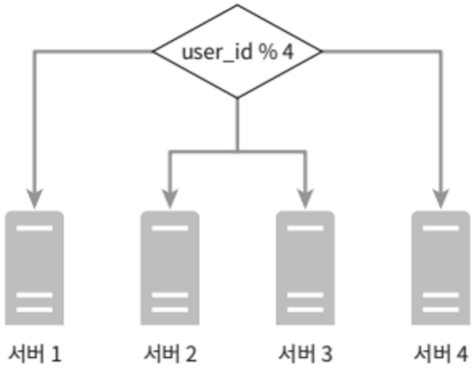

그림 15-5

가장 먼저 떠오르는 해결책은 데이터를 샤딩하여 여러 서버에 나누어 저장하는 것이다.

샤딩을 통해 급한 불은 껏다고 해도 문제의 근본적인 원인은 해결되지 못했다. 에어비앤비나 넷플릿는 S3를 쓴다고 하는데.. S3를 쓴다고 한다.

엥.. ?

책을 보면서 진지하게 이 부분에 대해서 의심을 하게 됐다.

구글 드라이브는 클라우드 저장소가 아닌가..? 왜 저장소를 만들면서 다른 회사의 클라우드 저장소를 사용하는 거지..? 좋은 서비스 구축이면 되는건가..?

뭐 아무튼 그렇다고 합니다...

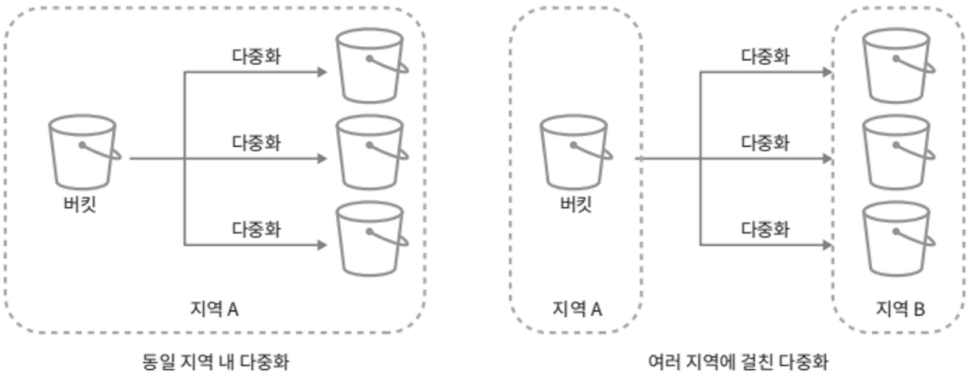

그림 15-6

S3 는 지원하는 기능도 많고 가용성도 높고 다중화도 지원하기 때문에 좋다고 한다. 그래서 위 같이 파일을 S3에 저장하고 보니, 저장소 문제도 해결 됐는데 개선할 부분이 있다고 한다.

- 로드 밸런서 : 역시 이번에도 빠지지 않고 나왔다.
- 웹 서버 : 로드 밸런서를 추가하면 더 많은 웹서버를 추가할 수 있다.
- 메타데이터 데이터베이스 : 데이터베이스를 파일 저장 서버에서 분리하여 SPOF를 회피한다. 데이터베이스도 또중화 및 또딩을 적용한다.
- 파일 저장소: S3를 파일 저장소로 사용하고, 가용성과 데이터 무손실을 보장 하기 위해서 두 개 이상의 지역에 데이터를 다중화 한다.

  ⇒ 이건 구글 드라이브 설계가 아니라 SaaS 아닌가?

뭐 대충 이런걸 완료하고 나면,

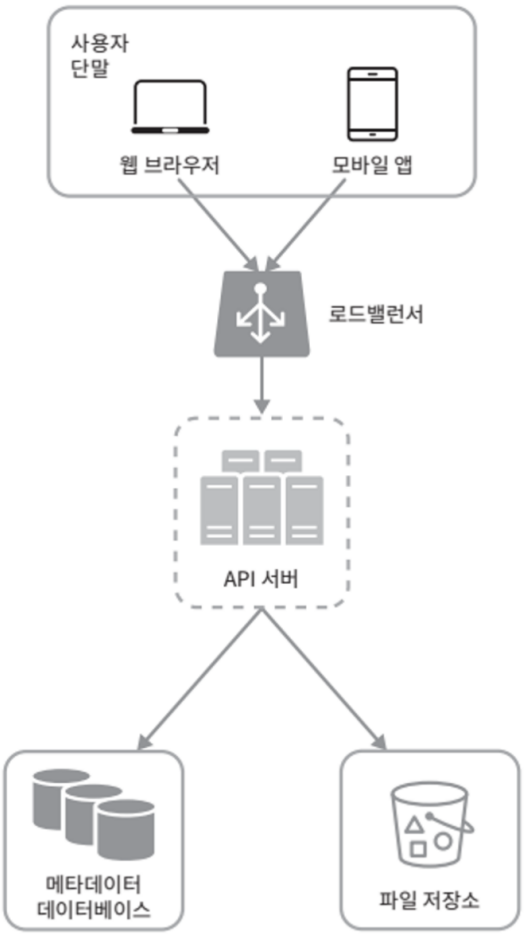

그림 15-7

뭐 대충 이런게 나온다고 한다.

동기화 충돌

두 명 이상의 사용자가 동시에 같은 파일이나 폴더를 업로드 하려고 하면 어떻게 되는 걸까?

사실 이런 충돌 전략을 사용하는 것은.. 뭐 S3 사용하면 알아서 완전관리형이니 해주지 않나? 생각이 들지만, 먼저 처리 되는 것은 변경된 것으로 보고 나중에 변경된 것은 충돌로 처리하는 것이다.

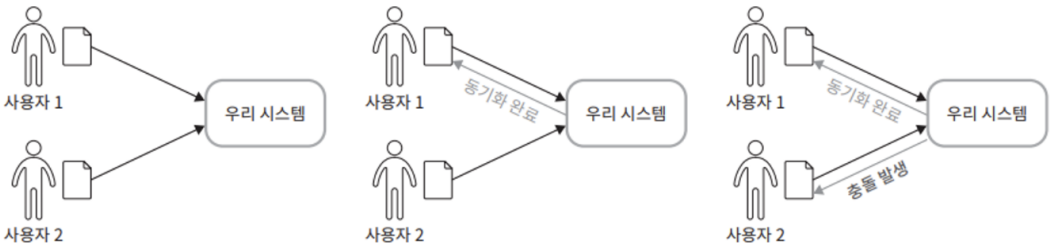

그림 15-8

뭐 2 버전이 생기는 건데, 어떻게 해결할 지는 책에 소개된 [4][5]내용 보라고 나와 있다.

해결책을 따로 알려주진 않는다.

저자도 15장은 쓰기 싫었던 것이 아닐까?

개략적 설계안

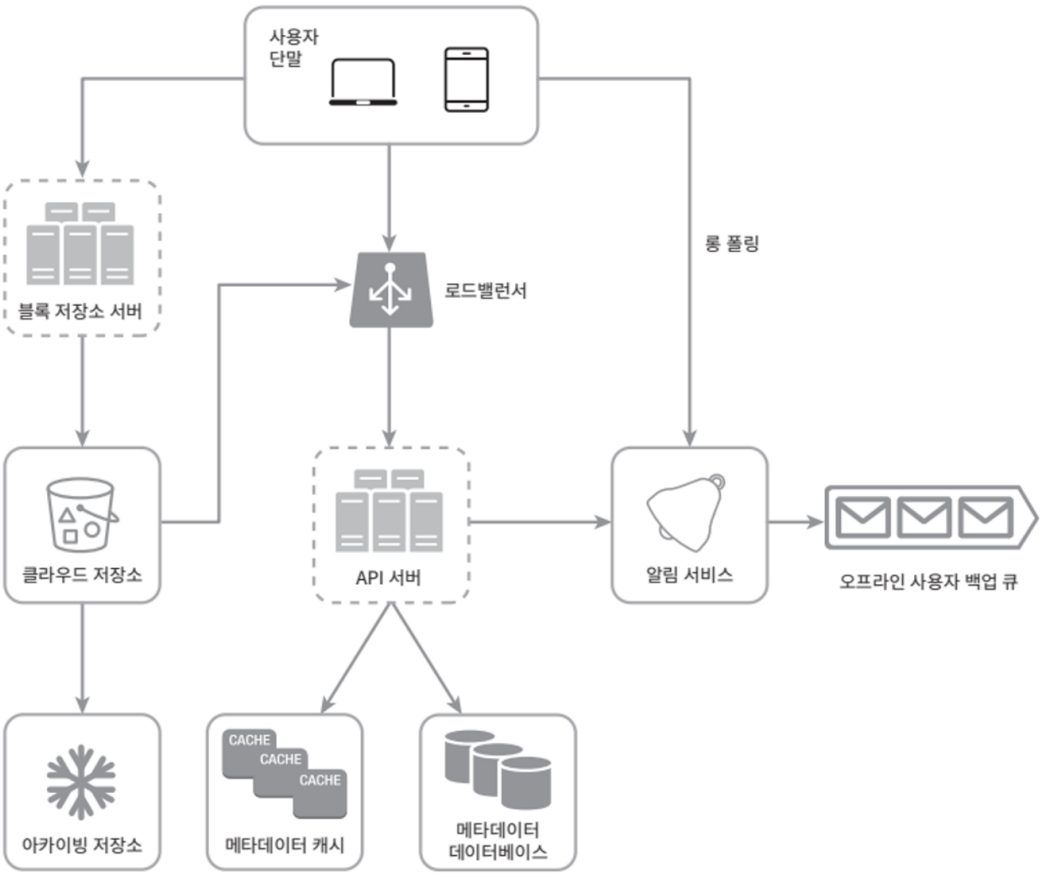

그림 15-10

대충 이런 컴포넌트들을 가진 설계안이 나온다고 한다. 개략적인 설계에서 사실 상당한 진행이 된 모습 단일 서버에서 출발 한다더니 여러 컴포넌트 가져와서 별다른 해결책도 없이 잔뜩 늘려놨다. 15장을 더 볼 필요가 있는 걸까? 뭐 아무튼,

각각의 컴포넌트를 소개하면

- 사용자 단말 : 클라이언트
- 블록 저장소 서버: 파일 블록을 클라우드 저장소에 업로드하는 서버다. 블록 저장소는 블록 수준 저장소라고도 불리며 클라우드 환경에서 데이터를 저장하는 기술이다. 이 저장소는 파일을 여러개의 블록으로 나누며 각 블록에는 고유 해시 값이 할당된다. 각 블록은 독립적인 객체로 취급된다고 하며 저장소에 저장된다고 한다. 여기서는 S3를 의미한다.

  ⇒ 이럴 거면 S3를 쓰지 클라우드 저장소를 따로 쓰는 의미가 있나?
- 클라우드 저장소 : 파일이 블록 단위로 나눠져 클라우드 저장소에 보관된다.
- 아카이빙 저장소 : 오랫동안 활성되지 않은 비활성 데이터를 저장하기 위한 컴퓨터 시스템이다.

- 로드밸런서 : 이하 생략
- API 서버 : 생략
- 메타데이터 베이스 : 생략
- 메타데이터 캐시 : 자주 쓰이는 데이터 캐시
- 알림 서비스
- 오프라인 사용자 백업 큐 : 클라이언트가 접속 중이 아니라서 파일의 최신 상태를 확인할 수 없을 때는 해당 정보를 이 큐에 두어 나중에 클라이언트가 접속 했을 때 동기화될 수 있도록 한다.

3단계 상세 설계

블록 저장소 서버

정기적으로 갱신되는 큰 파일들은 업데이트가 일어날 때마다 서버로 보내면 네트워크 대역폭을 많이 잡아먹는다고 한다. 최적화 방법으론 두 가지가 있다고 한다.

1. 델타 동기화 : 파일이 수정되면 전체 파일 대신 수정된 블록만 수정하는 방법이다.
2. 압축 : 블록 단위로 압축해두면 데이터 크기를 줄일 수 있다. 압축 알고리즘은 파일 유형에 따라 정해진다.

이 시스템에서 블록 저장소 서버는 파일 업로드에 관계 된 힘든 일을 처리하는 컴포넌트다. 클라가 보낸 파일을 블록 단위로 나누고, 각 블록에 압축 알고리즘을 적용한다. 전체 파일 중 수정된 블록만 전송하는 등 다양한 작업을 하고 다음과 같이 동작한다.

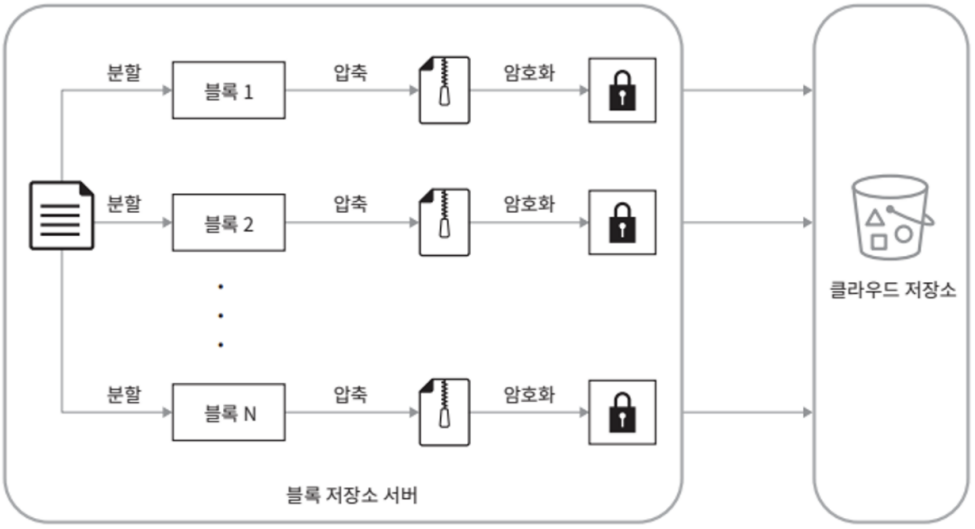

그림 15-11

1. 주어진 파일 블록 분할
2. 각 블록 압축
3. 클라우드 저장소로 보내기 전 암호화
4. 클라우드 저장소로 전송

높은 일관성 요구사항

이 시스템은 강한 일관성 모델을 기본적으로 지원해야 한다. 같은 파일이 다른 단말이나 사용자에 따라 다르게 보여선 안된다는 소리다. 메타데이터 캐시와 영속성 계층에도 같은 원칙이 적용되야 한다.

메모리 캐시는 보통 결과적 일관성 모델을 지원한다. 따라서 강한 일관성을 보장하기 위해선 다음 사항을 보장해야 한다.

- 캐시에 보관된 사본과 데이터베이스의 원본은 일치해야 한다.
- 원본에 변경이 발생하면 캐시를 무효화 한다.

RDB는 높은 일관성을 보장하지만 NoSQL은 그렇지 않기 때문에 여기선 RDBMS를 사용한다.

메타데이터 데이터베이스

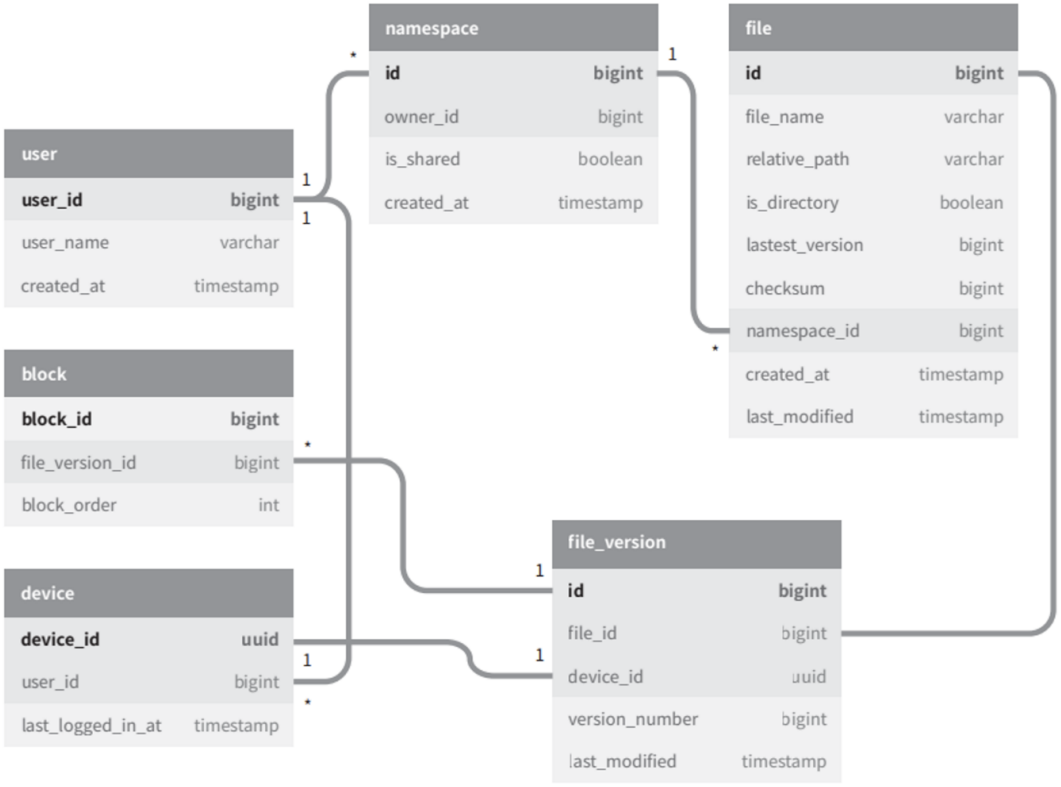

그림 15-13

이 시스템에서 사용할 주요 스키마다.

- user
- device ⇒ 단말 정보
- namespace ⇒ 사용자 루트 디렉터리 정보
- file ⇒ file 테이블의 최신 정보
- file_version ⇒ 파일의 갱신 이력
- block ⇒ 파일 블록에 대한 정보 보관

업로드 절차

자 이제 사용자가 업로드하면 어떻게 되는가?

- **파일 메타데이터 추가**

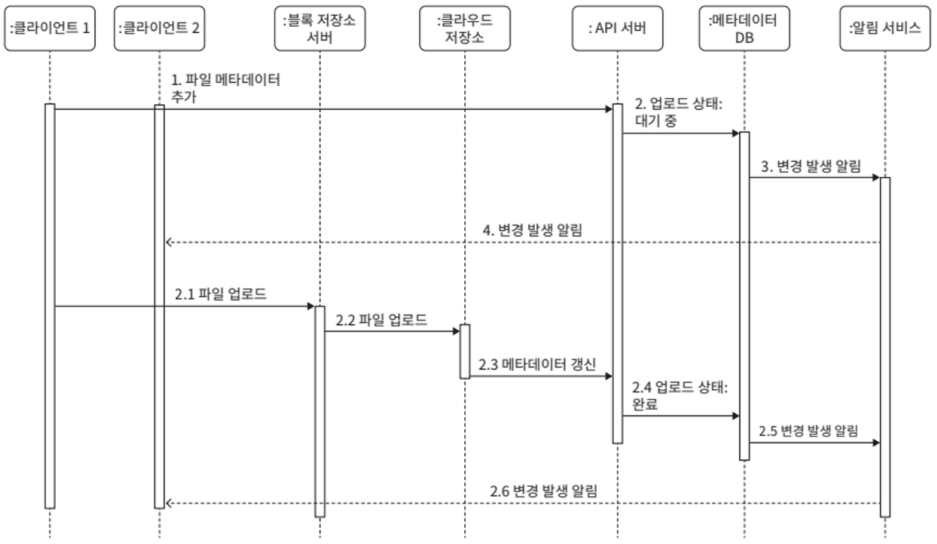

그림 15-14

1. 클라이언트 1이 새 파일의 메타데이터를 추가하기 위한 요청 전송
2. 새 파일의 메타데이터를 데이터베이스에 저장하고 업로드 상태를 대기중으로 변경
3. 새 파일이 추가 됐음을 알림 서비스에 통지
4. 알림 서비스는 관련된 클라이언트에게 파일이 업로드됨을 알림

- 파일을 클라우드 저장소에 업로드
  1. 클라1이 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, 이 콜백 호출은 API 서버로 전송됨
  4. 메타데이터 디비에 기록된 해당 파일의 상태를 완료로 변경
  5. 알림 서비스에 파일 업로드가 끝남을 통지
  6. 알림 발송

다운로드 절차

클라이언트가 다른 클라이언트가 파일을 편집했거나 추가했다는 사실을 어떻게 감지하는 걸까?

- 클라1이 접속 중이고 다른 클라가 파일을 변경하면 알림 서비스가 1에게 변경이 발생했으니 새 버전을 끌고 가야 한다고 알린다.
- 클라1이 접속 중이 아닐 때는 데이터가 캐시에 보관됐다가, 접속 중으로 바뀌면 클라1은 새 버전을 가져갈 것이다.

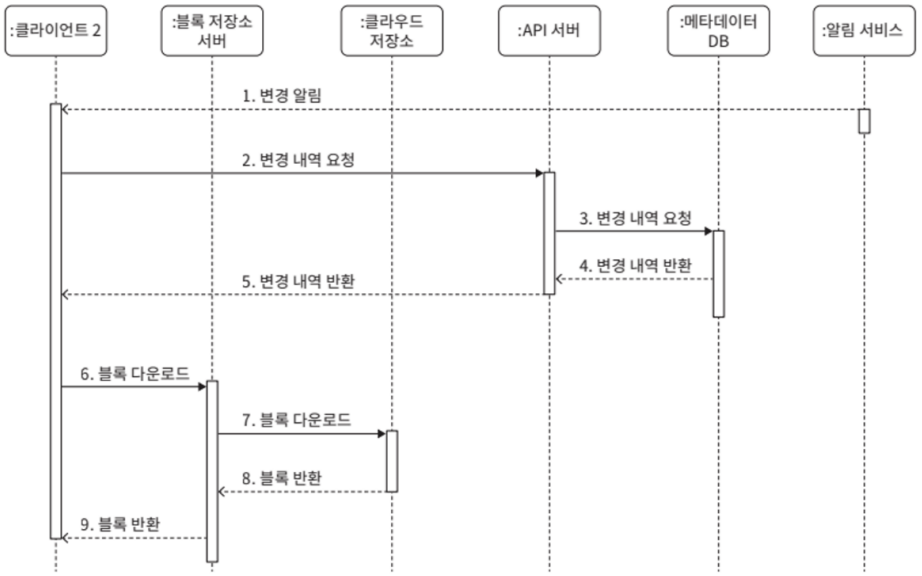

그림 15-15

파일이 변경됐음을 감지한 클라이언트는 우선 API 서버를 통해 메타데이터를 새로 가져가야 하고, 그 다음 블록들을 다운받아 파일에 재구성해야 한다.

1. 알림 서비스가 클라2에게 누군가가 파일을 변경했음을 알림
2. 알림 확인한 클라2는 새 메타데이터 요청
3. API 서버는 메타데이터 디비에 새 메타데이터 요청
4. 클라2에게 새 메타데이터가 반환됨
5. 클라2는 새 메타데이터를 받은 즉시 블록 다운로드 요청
6. 블록 저장소는 클라우드 저장소에 블록 다운로드
7. 클라우드 저장소는 블록 서버에 블록 요청 반환
8. 블록 저장소는 클라에게 요청 블록 반환, 클라2는 전송된 블록 사용하여 파일 재구성

알림 서비스

파일의 일관성을 유지하기 위해, 파일 변경이 있는 순간 다른 클라에게 알려 충돌 가능성을 줄여야 한다. 알림 서비스는 이 목적으로 사용된다. 단순하게 보면 이벤트 데이터를 클라로 보내는 서비스다. 두 가지 선택지가 있는데,

- 롱 폴링 : 드롭박스가 채택
- 웹 소켓 : 클라와 서버 사이 지속적 통신 채널을 제공한다.

근데 롱 폴링이 더 좋다. 알림을 그렇게 자주 보낼 필요도 없고 설정 비용도 적고 양방향 소통도 필요 없기 때문이다.

롱 폴링 방식을 쓰게 되면 각 클라이언트는 알림 서버와 롱 폴링용 연결을 유지하다 변경을 감지하면 해당 연결을 끊는다. 이 때 클라는 반드시 메타데이터 서버와 연결해 파일의 최신 내역을 다운로드 해야한다. 해당 다운로드 작업이 끝났거나 연결 타임아웃 시간이 도달한 경우, 즉시 새 요청을 보내어 롱 폴링 연결을 복원하고 유지해야 한다.

저장소 공간 절약

갱신 이력을 보존하고 안정성을 위해 파일의 여러 버전을 여러 데이터센터에 보관할 필요가 있는데, 저장용량이 너무 빠르게 소모될 수 있다.

- 중복 제거
- 지능적 백업 전략
  - 한도 설정 : 보관 파일 수
  - 중요 버전만 보관 : 자주 바뀌는 파일은 중요한 것만
- 자주 안쓰이는 건 아카이빙 저장소 보관

장애 처리

- 로드 밸런서 장애 : 부 로드 밸런서가 이어 받도록 설정하고, 심장 박동을 주기적으로 확인하는 헬스체크 옵션을 사용한다.
- 블록 서비스 서버 장애 : 다른 서버가 이어 받는다.

- 클라우드 저장소 장애 : S3는 다중화할 수 있기 때문에 다른 지역에서 파일을 가져오면 된다.
- API 서버 장애 : 트래픽 차단, 다른 서버가 이어 받는다.
- 메타데이터 캐시 장애 : 캐시도 다중화 하기 때문에 다른 노드가 이어 받는다.
- 메타데이터 베이스 장애
  - 주 서버 : 부 서버중 하나를 주로 바꾸고 주는 차단한다.
  - 부 서버 : 다른 부 서버가 대체하도록 하고 새롭게 대체한다.
- 알림 서비스 장애 : 롱 폴링 재생성
- 오프라인 사용자 백업 큐 장애 : 다중화

4단계 마무리

- 분할 압축 암호화 로직을 클라에 둬야 하기 때문에 플랫폼 별 구축 고려
- 클라가 해킹당할 수 있기 때문에 암호화 따로 빼서 사용하는 걸 고려