15장. 구글 드라이브 설계

발표자: 제이(소재훈)

목차

1. 서론

2. 1단계 - 문제 이해 및 설계 범위 확정

3. 2단계 - 개략적 설계안 제시 및 동의 구하기

4. 3단계 - 상세 설계

5. 4단계 - 마무리

1. 서론

DropBox

iCloud

Google Driver

클라우드 저장소

클라우드 저장소의 특징

1. 파일 저장 및 동기화 서비스 지원

2. 등록된 파일을 공유할 수 있는 서비스 지원

2. 1단계 - 문제 이해 및 설계 범위 확정

Q. 구글 드라이브와 같은 클라우드 저장소를 설계해보세요.

질문 1. 가장 중요하게 지원해야 하는 기능은 무엇인가요?

- 파일 업로드 / 다운로드, 파일 동기화, 알림 서비스

질문 2. 모바일 앱이나, 웹 앱 등의 지원 단말 여부

- 모바일 앱, 웹 앱 둘 다 지원

질문 3. 파일 암호화 지원 여부

- 파일 암호화 지원

2. 1단계 - 문제 이해 및 설계 범위 확정

Q. 구글 드라이브와 같은 클라우드 저장소를 설계해보세요.

질문 4. 파일 크기의 제한 여부

- 파일의 크기는 10GB 제한

질문 5. 서비스 사용자 수

- 일일 사용자 수(DAU)는 천만명

2. 1단계 - 문제 이해 및 설계 범위 확정

문제에 대한 설계 시 집중해야 할 요소들 정리

기능적 요소

1. 파일 추가 - 단순 구글 드라이브 안에 추가하는 방식
2. 파일 다운로드
3. 여러 단말 간 동기화 기능 지원
4. 파일 갱신 이력 조회 기능
5. 파일 공유 기능
6. 파일 편집 or 삭제, 공유 시 표시 기능 지원

비기능적 요소

1.데이터 손실이 발생되지 않도록 서비스 안정성 고려
2. 빠른 동기화 속도 지원
3. 모바일 사용자 위한 적절한 네트워크 대역폭의 소모량
4. 많은 양의 트래픽 처리가 가능한지 고려
5. 서비스에 일부 기능이 장애가 발생되어도 계속 사용가능한지 가용성을 고려

2. 1단계 - 문제 이해 및 설계 범위 확정

개략적인 설계 추정치 정리

1. 가입 사용자는 5천만명, 일일 사용자(DAU)는 천만명
2. 모든 사용자에게 10GB 무료 저장공간 제공
3. 매일 각 사용자가 평균 크기 500KB의 2개의 파일을 업로드한다고 가정
4. 읽기와 쓰기의 비율은 각각 1:1 비율

- 필요 저장공간 = 5천만명 * 10GB = 500페타 바이트
- 업로드 QPS = 천만명 * 2회 / 24시간 / 3600초 = 매 초 약 240번 QPS 발생
- 최대 QPS = 업로드 QPS * 2 = 480회

3. 2단계 - 개략적 설계안 제시 및 동의 구하기

- **서버 1대를 기준의 설계**

  필요 요소 (3가지)

1. 파일 업로드, 다운로드 등의 과정을 처리할 웹 서버
2. 사용자의 데이터 및 파일 정보를 저장할 데이터베이스
3. 파일을 저장할 저장소 시스템
   - 각 사용자별로 파일 시스템을 관리함

그림 15-3

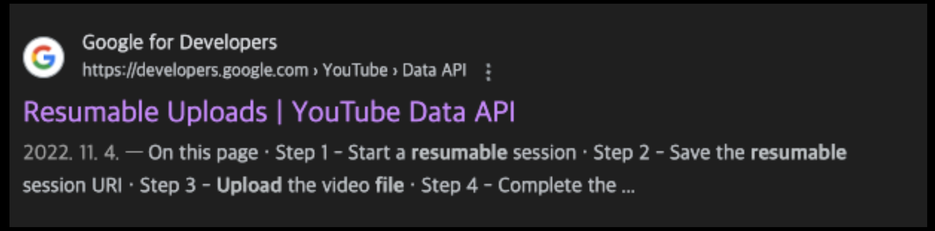
3. 2단계 - 개략적 설계안 제시 및 동의 구하기

API

- 모든 API는 사용자 인증 과정 수행 및 HTTPS 지원

1. 파일 업로드 API

- 파일의 크기가 작을때 - 단순 업로드
- 파일의 크기가 크고 중간에 중단될 수 있을 때 - 이어 올리기



3. 2단계 - 개략적 설계안 제시 및 동의 구하기

API

2. 파일 다운로드 API

- **Path: 다운로드할 파일의 경로**

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

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

3. 파일 갱신 히스토리 API

- **Path: 갱신 히스토리를 가져올 파일의 경로**
- **Limit: 히스토리 길이의 최대치**

```
https://api.example.com/files/list-revisions
```

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

3. 2단계 - 개략적 설계안 제시 및 동의 구하기

한 대 서버의 제약 극복

- 업로드되는 파일이 많아지면, 파일 시스템은 가득 차게됨

방법 1. user_id를 기준으로 샤딩하는 방법

-> 그러나 서버에 장애가 발생하면 데이터를 잃게됨

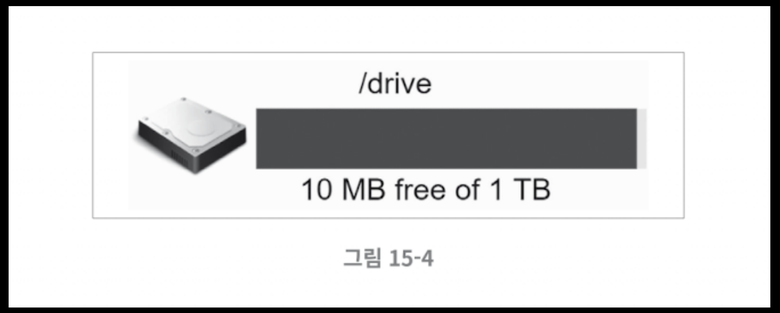


그림 15-4

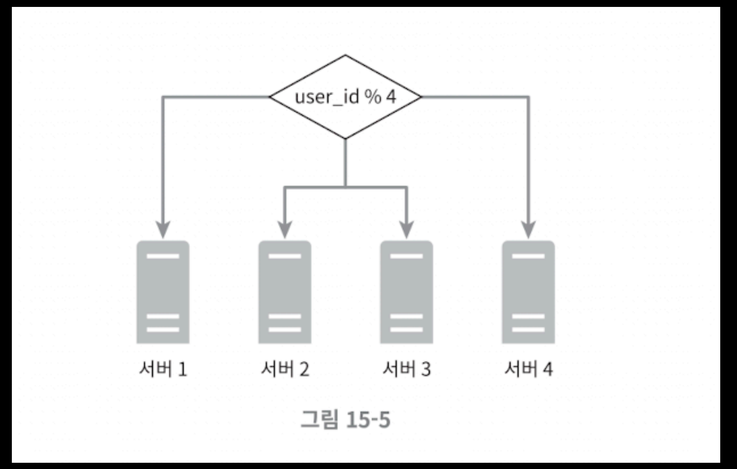


그림 15-5

3. 2단계 - 개략적 설계안 제시 및 동의 구하기

방법2. AWS S3 사용

- 같은 지역, 다른 지역을 걸쳐 다중화 처리가 가능

1. 동일 지역 내 다중화

2. 여러 지역 내 다중화 (추천)

- 데이터 손실 차단 및 가용성을 최대한 보장함

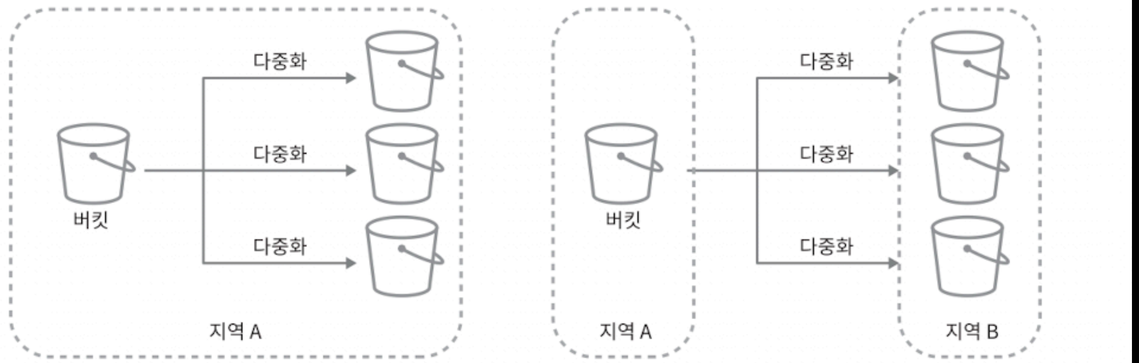


그림 15-6

3. 2단계 - 개략적 설계안 제시 및 동의 구하기

서버 1대에서 여러 대의 서버로 설계 개선

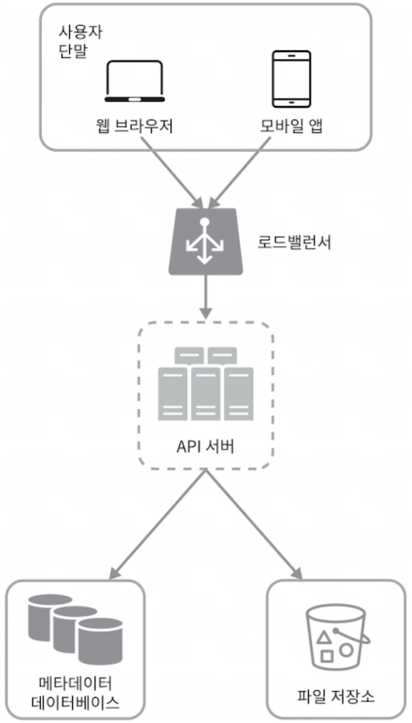


그림 15-7

1. 로드 밸런서 적용
2. 여러 대의 웹 서버
3. 다중화 및 샤딩 정책을 적용한 DB
4. 파일 손실을 방지하기 위한 여러 지역에 S3 다중화

3. 2단계 - 개략적 설계안 제시 및 동의 구하기

동기화 충돌

- 2명 이상의 사용자가 같은 파일이나 폴더를 동시에 업데이트 하는 경우 발생

**-> 깃허브처럼 git push 하기 이전에 깃허브 원격 저장소에 변동된 기록이 있다면,
해당 원격 저장소를 받아 온 후, 사용자가 변경한 파일을 추가하는 식으로 해결을 할 수 있음**

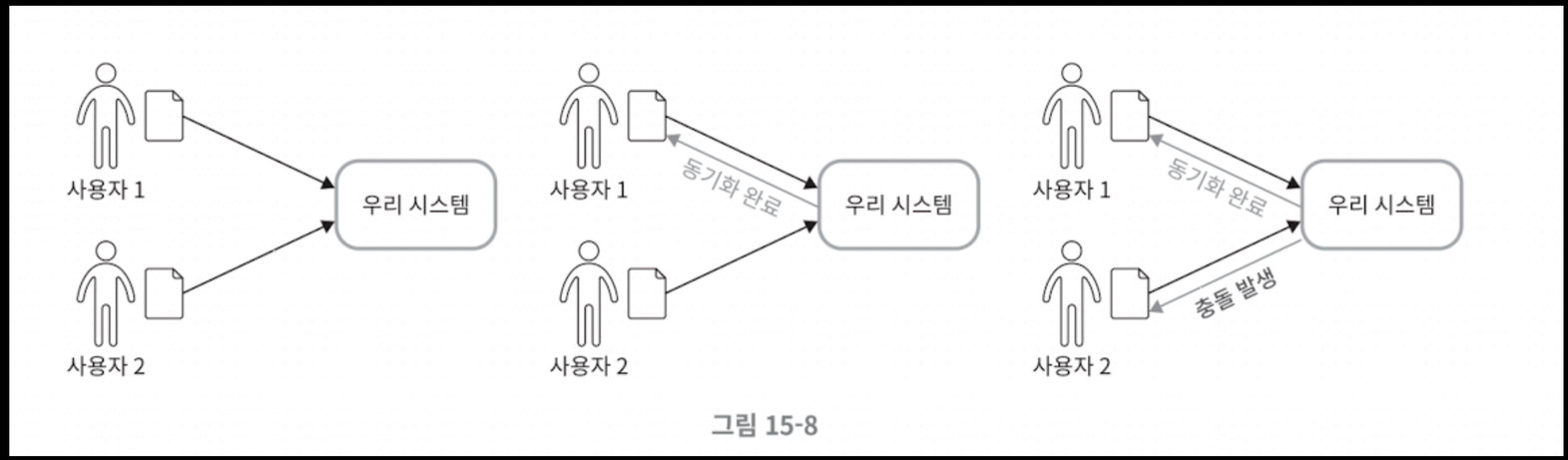


그림 15-8

3. 2단계 - 개략적 설계안 제시 및 동의 구하기

개략적인 설계안 정리

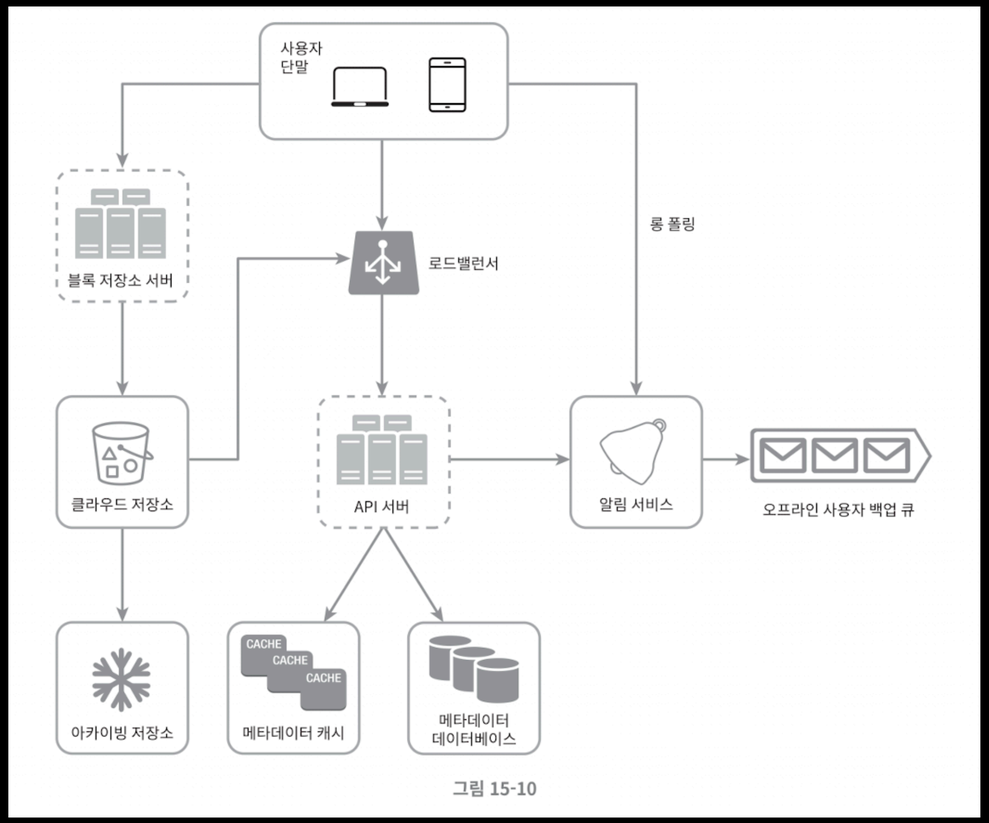


그림 15-10

1. 블록 저장소 서버

- 파일 블록을 클라우드 저장소에 업로드 하는 서버
- 클라우드 환경에서 데이터 파일을 저장하는 기술

2. 클라우드 저장소

- AWS S3와 같이 파일 블록을 저장하는 저장소

3. 아카이빙 저장소

- 오랫동안 사용하지 않은 비활성 데이터를 저장하는 저장소

4. 오프라인 사용자 백업 큐

- 접속 중이 아닐때, 해당 큐에 정보를 저장하고 접속 시 동기화함

4. 3단계 - 상세 설계

블록 저장소 서버

- 정기적으로 갱신되는 큰 파일들은 업데이트가 일어날 때마다, 전체 파일을 서버로 보내면
- 네트워크 대역폭을 많이 잡아 먹음

최적화 방법(2가지)

1. 델타 동기화: 파일 수정 시 전체 파일 대신, 수정이 일어난 블록만 동기화함

2. 압축: 블록 단위로 압축을 해 데이터 크기를 줄임

- 텍스트 파일 - gzip이나 bzip2를 사용
- 이미지, 비디오 파일 - 다른 압축 알고리즘 사용

4. 3단계 - 상세 설계

블록 저장소 서버 - 델타 동기화

- 수정이 일어난 블록 2, 블록 5만 갱신함

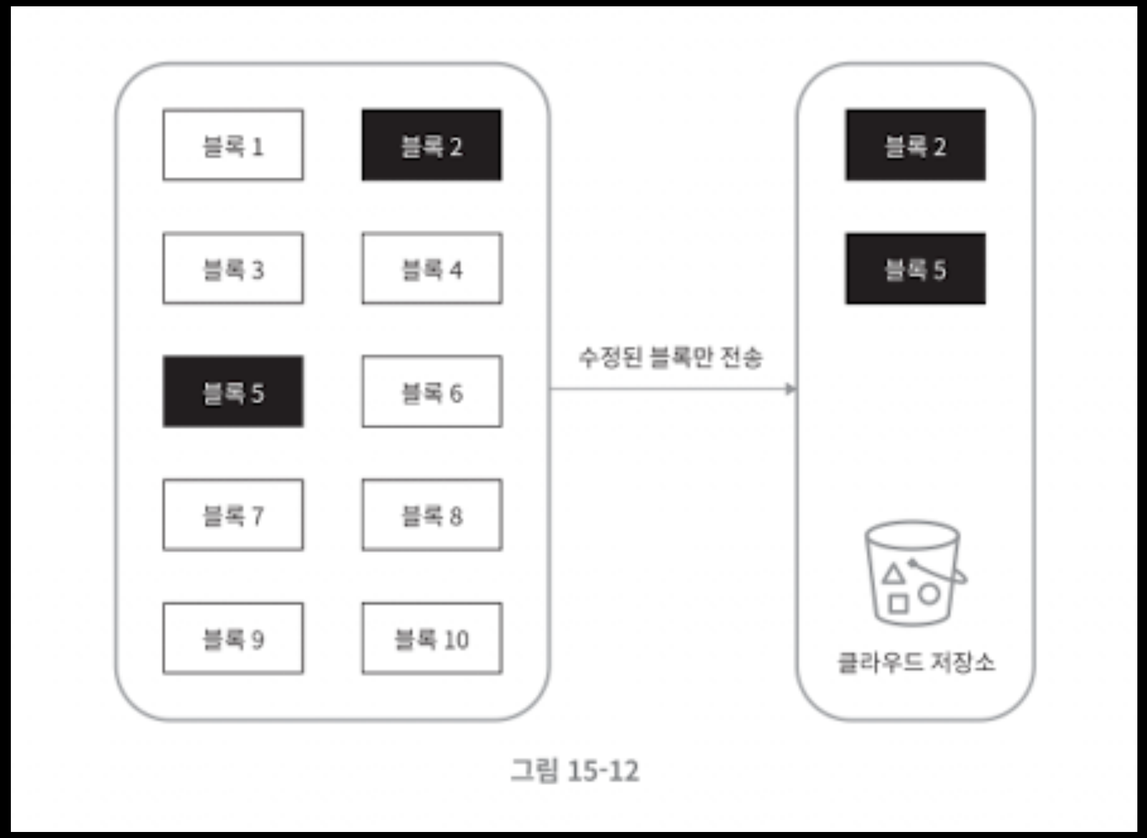


그림 15-12

4. 3단계 - 상세 설계

블록 저장소 서버 - 동작방식

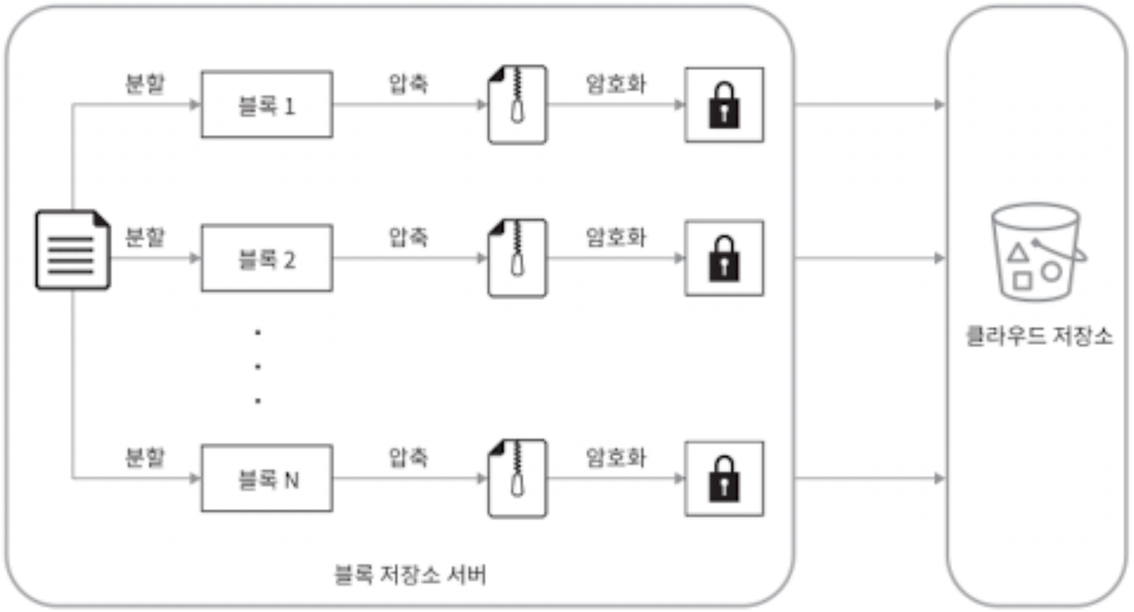


그림 15-11

1. 전송된 파일들을 작은 블록들로 분할함
2. 각 블록들을 압축함
3. 클라우드 저장소로 보내기 전 암호화
4. 암호화 한 후, 클라우드 저장소로 전송

4. 3단계 - 상세 설계

높은 일관성 요구사항

- 같은 파일이 단말기나 사용자에게 다르게 보이는 것을 허용하지 않는 강한 일관성 모델을 지원해야 함

-> 메모리 캐시의 경우 보통 최종 일관성을 보장함

-> NoSQL의 경우, ACID를 보장하지 않음

- 따라서 캐시에 보관된 사본과 데이터 베이스의 원본이 일치하도록 , DB 변경시 캐시에 있는 사본을 무효화
- NoSQL과 MySQL 사이의 동기화 로직을 작성함

-> 해당 설계에서는 RDBMS를 사용하여 일관성 요구사항에 대응함

4. 3단계 - 상세 설계

메타 데이터 데이터베이스

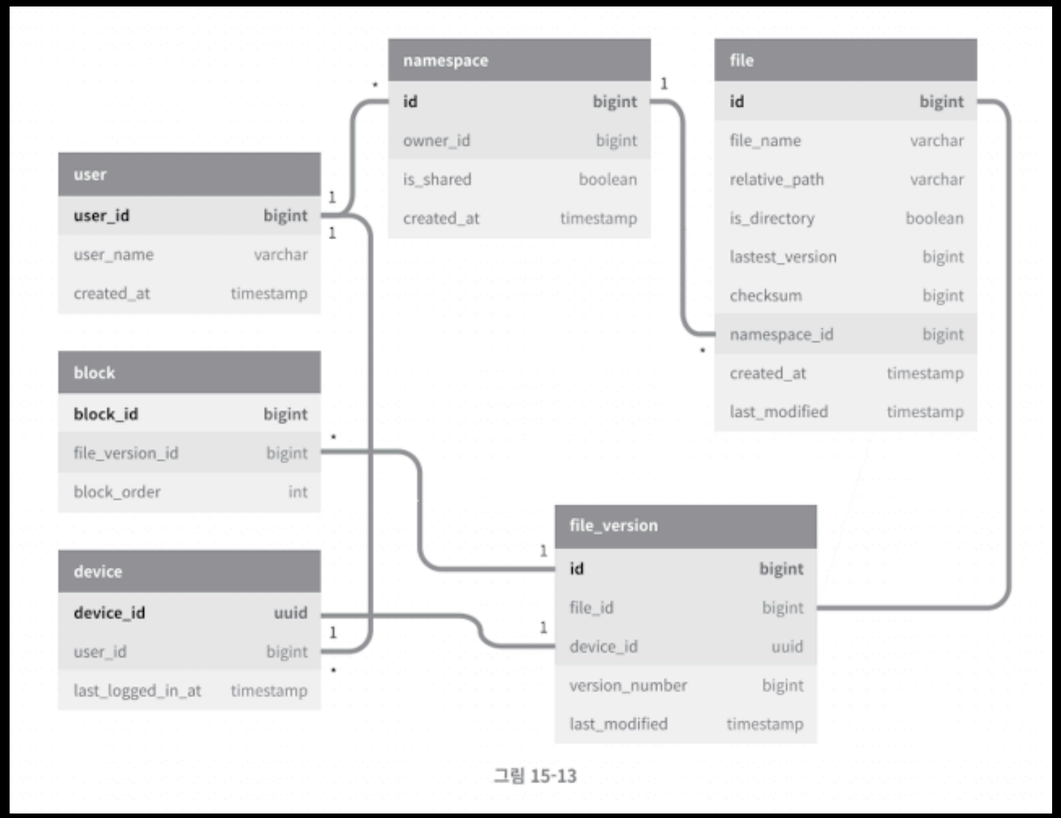


그림 15-13

User 테이블 - 사용자 정보를 저장

Device 테이블

- **device_id**
- 한 사용자가 여러 개의 단말을 가질 수 있어, **UUID형으로 설계함**

Namespace 테이블 - 사용자의 루트 디렉터리 정보 저장

File 테이블 - 파일의 최신 정보 저장

File_version 테이블(읽기 전용) - 파일의 갱신 이력 보관

Block 테이블 - 파일 블록에 대한 정보 보관

4. 3단계 - 상세 설계

업로드 절차

- 파일 수정의 경우에도 비슷하게 진행됨

Q. 2개의 업로드 절차가 병렬적으로 이루어지면 어떻게 되는가?

- 1번째 요청 - 메타 데이터 추가 요청
- 2번째 요청 - 파일을 클라우드 저장소로 업로드하기 위한 요청

메타 데이터 추가 요청

1. 클라이언트 1이 파일 메타 데이터 추가 요청

2. 새 파일의 메타데이터를 DB에 저장하고
업로드 상대를 대기 중으로 변경

3. 새 파일이 추가되었음을 알림 서비스에 통지함

4. 클라이언트2에게 파일 업로드가 되었다고 알림 서비스 전송

파일을 클라우드 저장소로 업로드하기 위한 요청

1. 클라이언트 1이 파일을 블록 저장소 서버에 업로드

2. 블록 저장소 서버는 블록 단위로 분할한다음, 압축과 암호화를 한 후
클라우드로 전송

3. 업로드가 끝나면 완료 콜백 호출, API서버로 전송됨

4. 메타데이터 DB에 기록된 해당 파일의 업로드가 끝났음을 알림 생성 후 전송

4. 3단계 - 상세 설계

업로드 절차

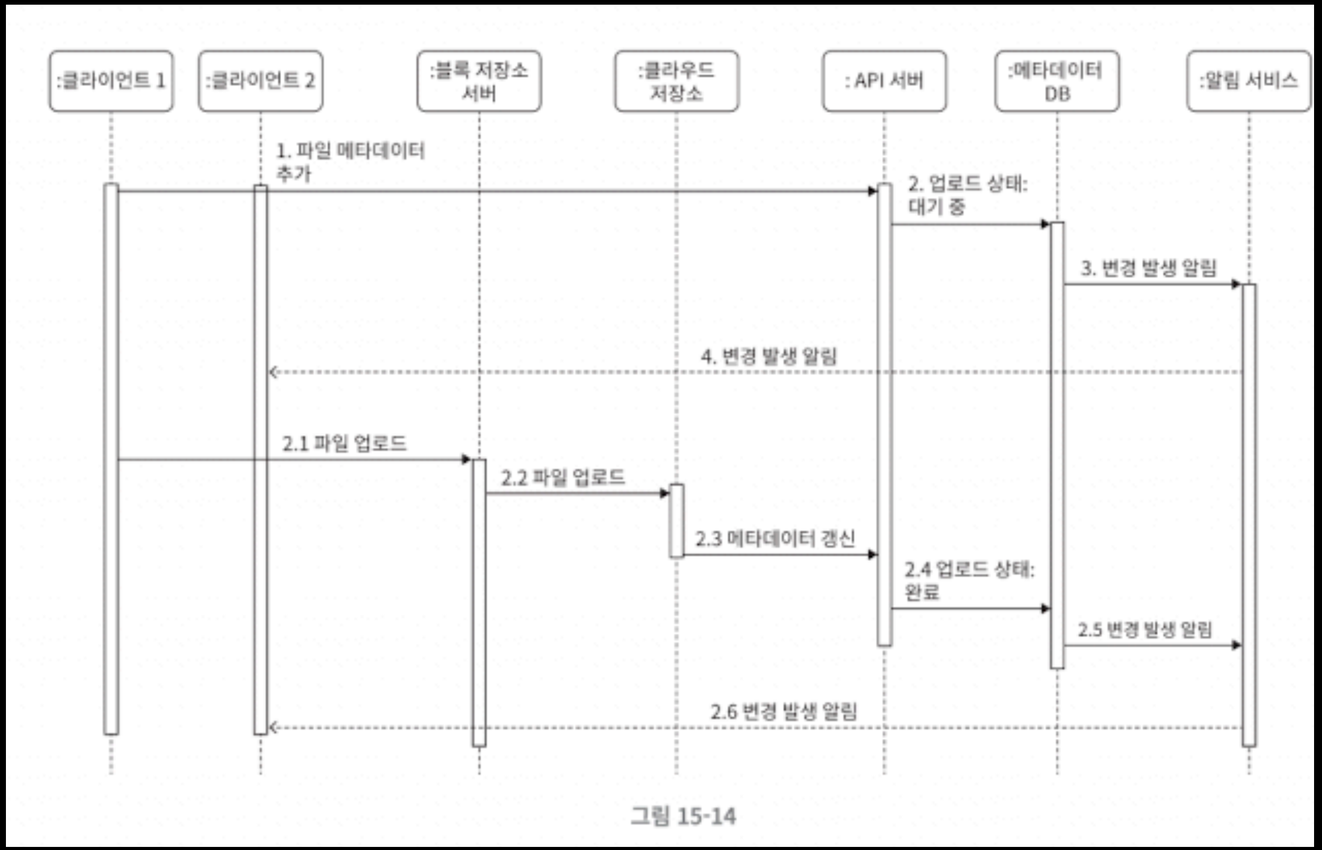


그림 15-14

4. 3단계 - 상세 설계

다운로드 절차

- 파일 다운로드는 파일이 새로 추가되거나 편집되면 자동으로 시작됨

Q. 클라이언트는 다른 클라이언트가 파일을 편집하거나 추가했다는 것을 어떻게 감지할까?

변경을 감지하는 방법(2가지)

1. 클라이언트 A가 접속 중이고, 다른 클라이언트가 파일 변경 시 알림 서버가 클라이언트 A에게 파일이 변경되었다고 알려줘야 함

2. 오프라인의 경우, 캐시에 임시 저장한 후 접속 시 변경된 내용을 가져와야 함

4. 3단계 - 상세 설계

다운로드 절차

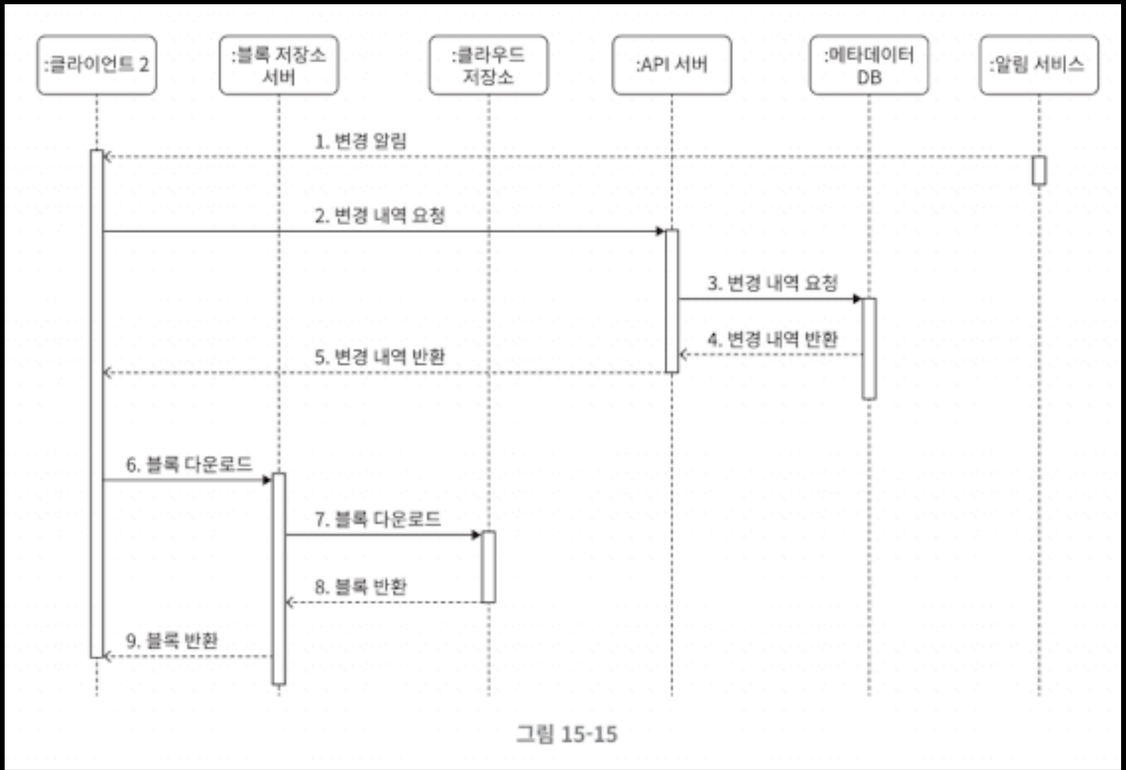


그림 15-15

1. 알림 서비스가 클라이언트2에게 누군가 파일을 변경했음을 알림
2. 알림을 확인한 클라이언트2는 새로운 메타 데이터를 요청
3. API 서버는 메타 데이터 DB에게 새로운 메타 데이터 요청
4. API 서버는 새로운 메타 데이터가 반환됨
5. 클라이언트 2에게 새로운 메타 데이터를 반환됨
6. 클라이언트 2는 새 메타 데이터를 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

- 클라이언트 2는 전송된 블록을 사용하여 파일 재구성

4. 3단계 - 상세 설계

알림 서비스

- 파일 일관성을 유지하기 위해, 클라이언트는 로컬에서 파일이 수정됨을 감지하는 순간 다른 클라이언트에게 알려서 충돌을 줄임

알림 서비스의 방법(2가지)

1. 롱 폴링: 드롭박스가 해당 방식 사용함

2. 웹 소켓: 클라이언트와 서버 사이에 지속적인 통신 채널을 제공함(양방향 통신)

-> 서버에서 변경이 된 사실을 클라이언트에게 알려주어야 하지만, 반대는 사용되지 않음(단방향 통신)

-> 단 시간에 많은 알림 데이터를 보낼 일은 없음

-> 롱 폴링 방식 선택함

4. 3단계 - 상세 설계

알림 서비스

- 각 클라이언트는 알림 서버와 롱 폴링용 연결을 유지하다가, 특정 파일에 대한 변경 감시지 해당 연결을 끊음

- 이때 클라이언트는 반드시 메타 데이터 서버와 연결해 파일의 최신 내역을 다운로드해야 함

-> 해당 다운로드 작업이 끝났거나 연결 타임 아웃 시간에 도달한 경우에
즉시 새 요청을 요청하여 롱 폴링 연결을 복원하고 유지해야 함

4. 3단계 - 상세 설계

저장소 공간 절약

- 파일 갱신 이력을 보존하고 안정성을 보장하기 위해서는 파일의 여러 버전을 여러 데이터센터에 보관할 필요가 있음

Q. 모든 버전을 자주 백업하면 저장용량이 너무 빨리 소진될 경우, 어떻게 대처하는가?

저장소 공간의 절약 방법(3가지)

1. 중복 제거: 중복된 파일 블록을 계정차원에서 제거함
   - 두 블록이 같은 블록인지 해시 값을 비교해서 판단함
2. 한도 설정 및 중요 버전만 보관하는 방식의 백업 전략을 채택함
   - 한도 설정: 보관해야 하는 파일 버전 개수에 상한을 둠 - 상한에 도달하면 가장 오래된 버전 버림
   - 중요 버전만 보관하는 방식: 업데이트가 짧은 시간 이루어지는 버전의 경우 중요한 버전만 저장함
3. 자주 사용되지 않는 데이터는 아카이빙 저장소로 옮김
   - 몇달 혹은 수년간 사용되지 않을 경우 아마존 S3 글래시어 같은 아카이빙 저장소로 옮김

4. 3단계 - 상세 설계

장애 처리

Q. 발생할 수 있는 장애의 유형으로는 무엇이 있는가?

1. 로드 밸런서 장애

- 로드밸런서에 장애 발생 시 부 로드밸런서가 활성화되어 트래픽을 이어 받아 처리하도록 개선

2. 블록 저장소 서버 장애

- 블록 저장소에 장애가 발생되었다면, 다른 서버가 미완료 상태 또는 대기 상태인 작업을 이어받아야 함

3. 클라우드 저장소 장애

- 여러 지역에 다중화된 S3의 경우, 다른 지역에서 파일을 가져오면 됨

4. 3단계 - 상세 설계

장애 처리

Q. 발생할 수 있는 장애의 유형으로는 무엇이 있는가?

4. API 서버 장애

- 로드밸런서는 API 서버에 장애가 발생시, 트래픽을 해당 서버에 보내지 않음으로 장애 서버를 격리함

5. 메타데이터 캐시 장애

- 메타데이터 캐시 서버의 다중화를 이용하여, 장애 발생 시 다른 노드에서 데이터를 가져옴

6.메타데이터 베이스 장애

- Master 서버 에러 - Slave 서버 중 하나를 Master로 승격시키고, 또다른 Slave노드 생성
- Slave 서버 에러 - 다른 Slave 서버를 사용하며, 또다른 Slave노드 생성

4. 3단계 - 상세 설계

장애 처리

Q. 발생할 수 있는 장애의 유형으로는 무엇이 있는가?

7. 알림 서비스 장애

- 접속 중인 모든 사용자는 알림 서버와 롱 폴링 연결을 하나씩 유지하며, 수 많은 사용자가 이용할 수록 롱 폴링 연결의 수는 늘어남

-> 장애 발생 시 롱 폴링 연결을 복구하는 것은 상대적으로 느릴 수 있음

8. 오프라인 사용자 백업 큐 장애

- 큐 또한 다중화하여 큐에 장애 발생 시 클라이언트들은 백업 큐로 구독 관계를 재설정해야 함

5. 4단계 - 마무리

정리

- 설계안은 1. 파일의 메타 데이터를 관리하는 부분과 2. 파일 동기화를 처리하는 부분으로 구분됨
- 이와 동시에 알림 서비스와 롱 폴링을 방식을 사용하여 파일 상태를 최신으로 유지할 수 있도록 구현됨
- 회사마다 요구하는 제약 조건에 따라 달라짐으로 면접관의 대화를 통해 설계 요소를 잘 이끌어내는 것이 중요

추가적으로 논의하면 좋은것

블록 저장소 서버를 거치지 않고 파일을 클라우드 저장소에 직접 업로드하는 경우

- 업로드 시간은 빨라짐
- 플랫폼 별로 분할, 압축, 암호화 로직을 클라이언트에 별도로 구현해야 함
- 클라이언트의 해킹 당할 가능성이 있으므로, 암호화 로직을 클라이언트 안에 두는 것은 적절하지 않을 수 있음

5. 4단계 - 마무리

추가적으로 논의하면 좋은것

- 접속 상태를 관리하는 로직을 별도 서비스로 옮겨, 다른 서비스에서 쉽게 활용할 수 있게 함

END