**DESIGN Google Drive**

Google Drive is a file storage and synchronization service that helps you store documents, photos, videos, and other files in the cloud. You can access your files from any computer, smartphone, and tablet.

### Understanding Problem and Establishing Design Scope:

Query1: What are the most important features?

-> Upload and download files, file sync, notification.

Query2: What is the DAU traffic?

-> 10 million Daily active users

Query3: Is this a mobile app, a web app, or both?

-> Both

Query5: What type of files are supported?

-> All file type

Query6: Is there a file size limit?

-> Yes, files must be 10GB or smaller

Query7: Do files need to be encrypted?

-> Yes, files must be encrypted.

Focusing on following features:

* Add files: The easiest way to add a file is to drag a file into Google drive.
* Download files.
* Sync files across multiple devices: When a file is added to one device, it is automatically synced to other devices.
* See file revisions.
* Share files with your friends, family, and coworkers.
* Send a notification when a file is edited, deleted or shared with you.

Non functional requirements:

* Reliability: Extremely important for a storage system. Data loss is unacceptable.
* Fast sync speed: If file sync takes too much time, users will become impatient and abandon the product.
* Bandwidth usage: If a product takes a lot of unnecessary network bandwidth, users will be unhappy, especially when they are on a mobile data plan.
* Scalability: The system should be able to handle high volumes of traffic.
* High availability: Users should still be able to use the system when some servers are offline, slowed down or have unexpected network errors.

### Back of the Envelope Estimation:

Assuming 50 million signed up users and

10 million DAU = 10^7 DAU

Assuming average file uploads is 2 files/day of on an average 500 KB

1:1 average read to write ratio

Total space needed = 50 \* 10^6 \* 10GB = 5 \* 10^8 \* 10^9 = 5\*10^17 = 500 PB

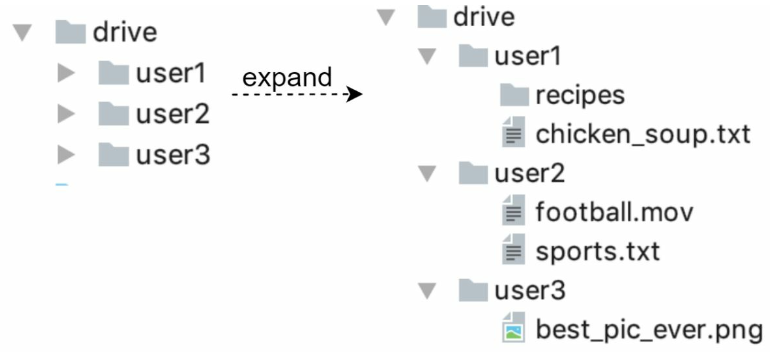
QPS for upload = 10^7 \**2 / (24\**60\*60) = 10^7\*2/86400 = 2\*10^7/10^5 = 200(approx)

Peak QPS = 400

### High Level Design Propositions and approaches:

Let us start with a single server setup as listed below:

1. A web server to upload and download files.
2. A database to keep track of metadata like user data, login info, files info, etc.
3. A storage system to store files. We allocate 1TB of storage space to store files.



**APIs**

We primarily need 3 APIs: upload a file, download a file and get a file version.

1. **Upload a file to Google Drive:**

Two types of uploads are supported:

* Simple upload: Use this upload type when the file size is small.
* Resumable upload: Use this upload type when file size is large and there is high chance of network interruption

Here is an example of resumable upload API:

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

Params:  
 • uploadType=resumable

• data: Local file to be uploaded.

A resumable upload is achieved by the following 3 steps [2]:   
 • Send the initial request to retrieve the resumable URL.  
 • Upload the data and keep track of the uploaded datastream.  
 • If the upload is disturbed, resume the upload for the datastream which is not uploaded

1. **Download a file from Google Drive**

Example API: <https://api.example.com/files/download>

Params:

Path: download file path

Example:

{"path": "/recipes/soup/best\_soup.txt" }

1. **Get file revisions:**

Example API: <https://api.example.com/files/list_revisions>

Params:  
 • path: The path to the file you want to get the revision history.

• limit: The maximum number of revisions to return.

Example params:

{

"path": "/recipes/soup/best\_soup.txt",

"limit": 20

}

All the APIs require user authentication and use HTTPs. Secure Sockets Layer(SSL) protects data transfer between the client and backend servers.

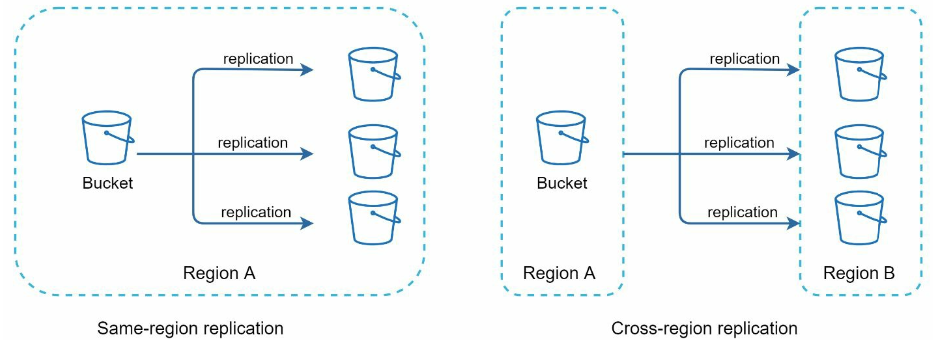
**Move away from single server**

Only 10 MB of storage space is left! This is an emergency as users cannot upload files anymore. The first solution comes to mind is to shard the data, so it is stored on multiple storage servers

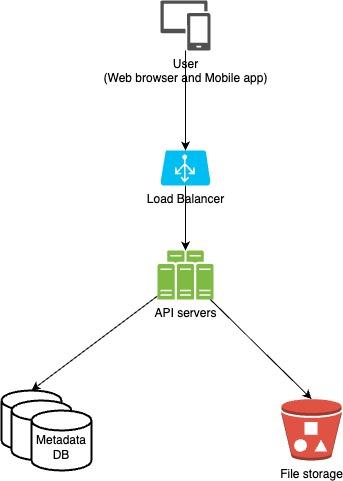
You have stopped the fire, but you are still worried about potential data losses in case of storage server outage. You ask around and your backend guru friend Frank told you that many leading companies like Netflix and Airbnb use Amazon S3 for storage.

**Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security and performance”**

Amazon S3 supports same-region and cross-region. A region is a geographic area where Amazon web services(AWS) have data centers.

Data can be replicated on the same-region(left side) and cross-region(right side).  
Redundant files are stored in multiple regions to guard against data loss and ensure availability. 

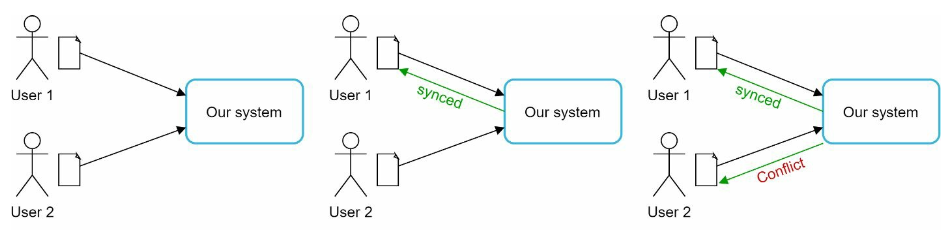
Other areas to improve:

* Load Balancer: Add a load balancer to distribute network traffic. A load balancer ensures evenly distributed traffic, and if a server goes down, it will redistribute the traffic.
* Web servers: After a load balancer is added, more web servers can be added/removed easily, depending on traffic load.
* Metadata database: Move the database out of the server to avoid a single point of Failure. In the meantime, set up data replication and sharding to meet the availability and scalability requirements.
* File storage: Amazon S3 is used for file storage. To ensure availability and durability, files are replicated into geographical regions.
* 

**Sync conflicts**For a large system like Google Drive, sync conflicts happen from time to time. When two users modify the same file or folder at the same time, a conflict happens.

How can we resolve the conflict?

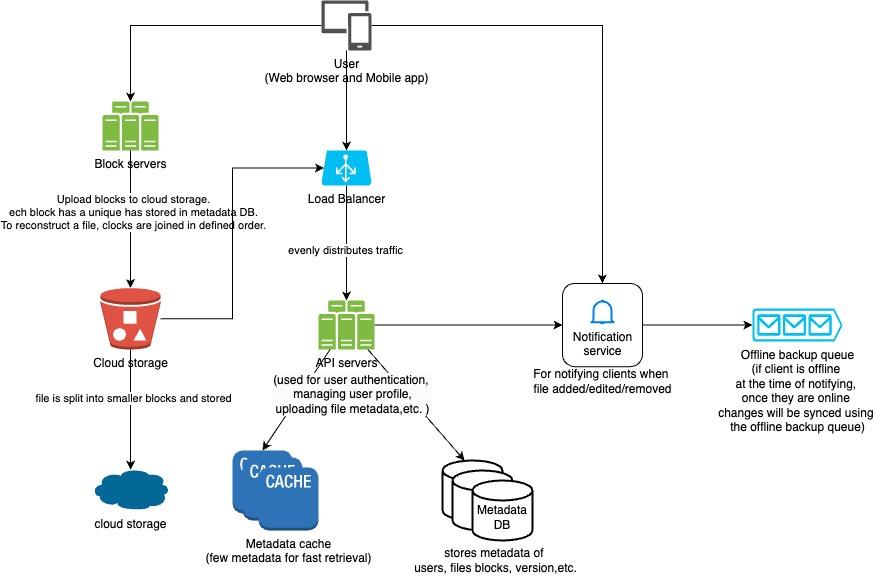
-> the first version that gets processed wins and the version that gets processed later receives a conflict.



How can we resolve the conflict for user 2?  
Our system presents both copies of the same file: user 2 ’s local copy and the latest version from the server. User 2 has the option to merge both files or override one version with the other.

While multiple users are editing the same document at the same time, it is challenging to keep the document synchronized.

**High level design**

****

### Design Deep Dive:

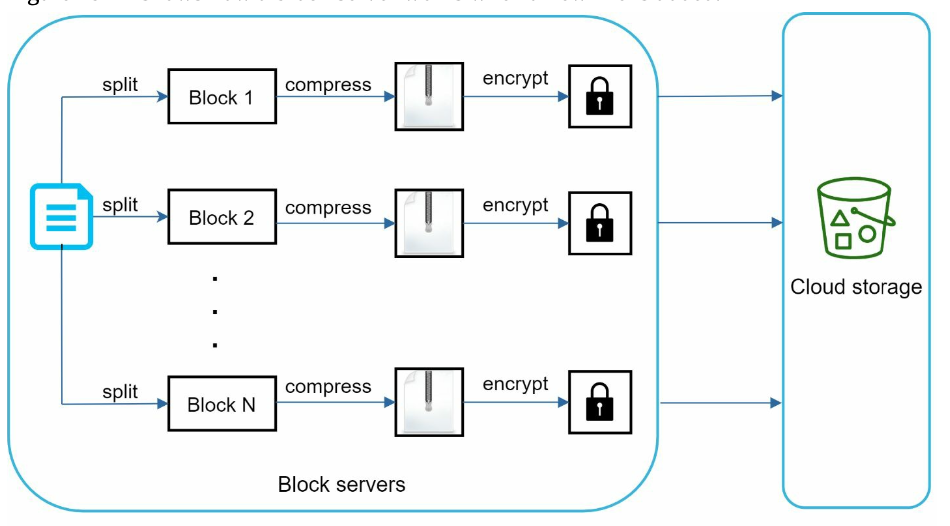
Block servers:

For large files that are uploaded regularly, sending the whole file on each update consumes a lot of bandwidth. Two optimizations are proposed to minimize the amount of network traffic being transmitted:

* Delta sync: When a file is modified, only modified blocks are synced instead of the whole file using a sync algorithm.
* Compression: Applying compression on blocks can significantly reduce the data size. Thus blocks are compressed using compression algorithms depending on file types. For example, gzip and bzip are used for compressing text files. Different compression algorithms are needed to compress images and videos.

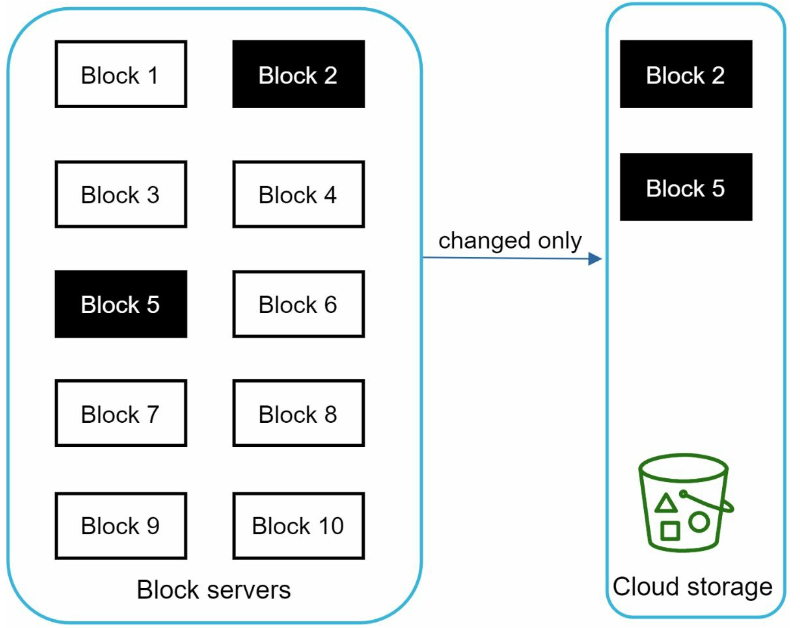
In our system block servers do the heavy lifting work for uploading files. Block servers process files passed from the clients by splitting a file into blocks, compressing each block, and encrypting them.

Instead of uploading the whole file to the storage only the modified blocks are transferred.



* A file is split into smaller blocks
* Each block is then compressed using compression algo in parallel.
* To ensure security, each block is also encrypted before it is sent to cloud storage in parallel.
* Blocks are uploaded to the cloud storage in parallel.

Delta sync, meaning only modified blocks are transferred to cloud storage. Highlighted blocks “block 2” and “block 5” represent changed blocks. Using delta sync, only those two blocks are uploaded to the cloud storage.



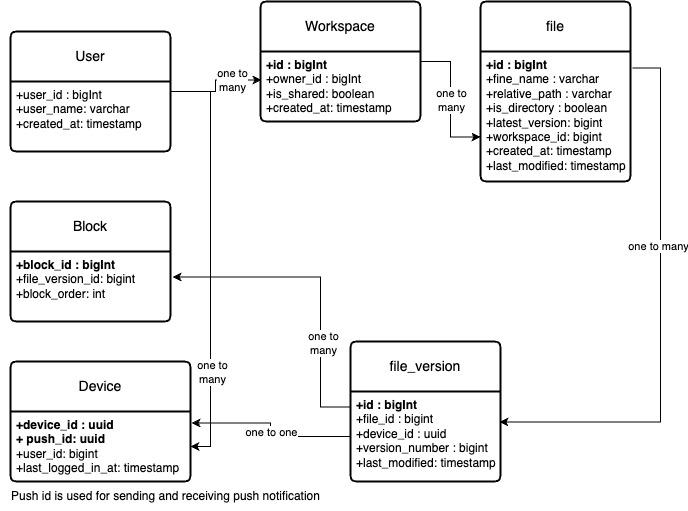
Block servers allow us to save network traffic by providing delta sync and compression.

**High consistency requirement**

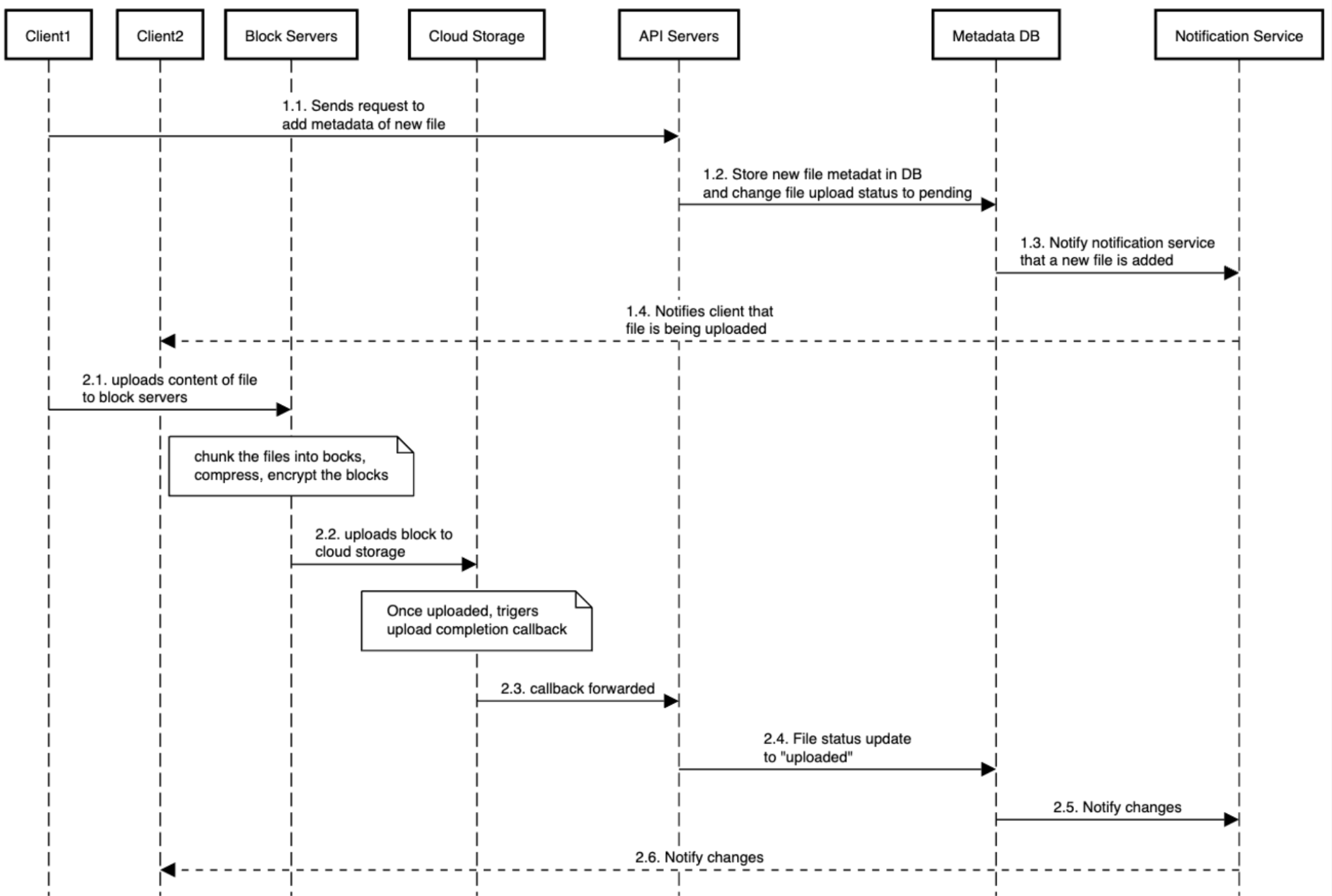
Our system requires strong consistency by default. It is unacceptable for a file to be shown differently for different clients at the same time. The system needs to provide strong consistency for metadata cache and database layers.  
Memory caches adopt an eventual consistency model by default, which means different replicas might have different data.

* Data in each replica and the master must be consistent.
* Invalidate caches on database write to ensure cache and database hold the same value.
* Achieving strong consistency in a relational database is easy because it maintains the ACID properties. However, NoSQL does not support ACID properties by default.

**Metadata Database**



**Upload Flow**

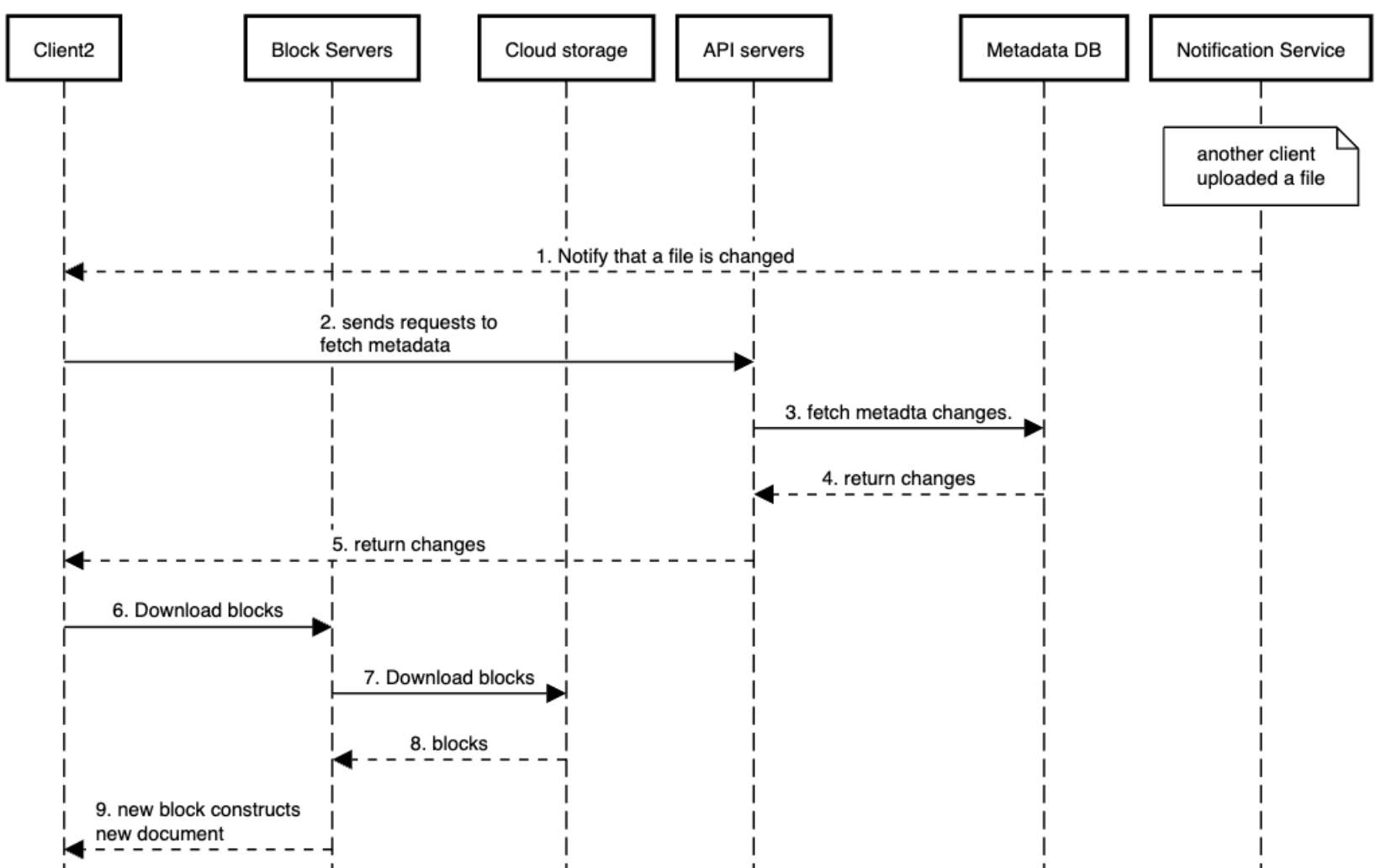


**Download Flow:**

Download flow is triggered when a file is added or edited elsewhere. The =re are two ways a client can know:

* If clientA is online while a file is changed by another client, notification service will inform client A that changes are made somewhere so it needs to pull the latest data.
* If clientA is offline while a file is changed by another client, data will be saved to the cache. When the offline client is online again, it pulls the latest changes.

Once a client knows a file is changed, it first requests metadata via API servers, then downloads blocks to construct the file.



**Notification service**

To maintain file consistency, any mutation of a file performed locally needs to be informed to other clients to reduce conflicts. Notification service is built to serve this purpose. At the high-level, notification service allows data to be transferred to clients as events happen. Here are a few options:

• Long polling. Dropbox uses long polling.

• WebSocket. WebSocket provides a persistent connection between the client and the server. Communication is bi-directional.

Even though both options work well, we opt for long polling for the following two reasons:

* Communication for notification service is not bidirectional. The server sends information about file changes to the client, but not vice versa.
* Websocket is suited for real-time bi-directional communication such as a chat app. For Google Drive Notifications are sent infrequently with no burst of data.

With long polling, each client establishes a long polling to the notification service. If changes to a file are detected, the client will close the long polling connection. Closing the connection means a client must connect to the metadata server to download the latest changes. After a response is received or connection timeout is reached, a client immediately sends a new request to keep the connection open.

**Save Storage space**

To support file version history and ensure reliability, multiple versions of the same file are stored across multiple data centers. Storage space can be filled up quickly with frequent backups of all file revisions. Three techniques are proposed to reduce storage costs:

* De-duplicate data blocks: Eliminating redundant blocks at the account level is an easy way to save space. Two blocks are identical if they have the same hash values.
* Adopt an intelligent data backup strategy:

1. Set a limit: Set limit for the number of versions to store. If the limit is reached, the oldest version will be replaced with the newer version.

2. Keep valuable versions only: Some files might be modified frequently. For example, saving every edited version for a heavily modified document could mean the file is saved over 1000 times within a short period. To avoid unnecessary copies, we could limit the number of saved versions. We give more weight to recent versions. Experimentation is helpful to figure out the optimal number of versions to save.

* Moving infrequently used data to cold storage. Cold data is the data that has not been active for months or years. Cold storage like Amazon S3 glacier is much cheaper than S3.

**Failure Handling**

Your interviewer might be interested in hearing about how you handle the following system failures:

• Load balancer failure: If a load balancer fails, the secondary would become active and pick up the traffic. Load balancers usually monitor each other using a heartbeat, a periodic signal sent between load balancers. A load balancer is considered as failed if it has not sent a heartbeat for some time.

• Block server failure: If a block server fails, other servers pick up unfinished or pending jobs.

• Cloud storage failure: S3 buckets are replicated multiple times in different regions. If files are not available in one region, they can be fetched from different regions.

• API server failure: It is a stateless service. If an API server fails, the traffic is redirected to other API servers by a load balancer.

• Metadata cache failure: Metadata cache servers are replicated multiple times. If one node goes down, you can still access other nodes to fetch data. We will bring up a new cache server to replace the failed one.

• Metadata DB failure.

* Master down: If the master is down, promote one of the slaves to act as a new master and bring up a new slave node.
* Slave down: If a slave is down, you can use another slave for read operations and bring another database server to replace the failed one.

• Notification service failure: Every online user keeps a long poll connection with the notification server. Thus, each notification server is connected with many users. According to the Dropbox talk in 2012 [6], over 1 million connections are open per machine. If a server goes down, all the long poll connections are lost so clients must reconnect to a different server. Even though one server can keep many open connections, it cannot reconnect all the lost connections at once. Reconnecting with all the lost clients is a relatively slow process.

• Offline backup queue failure: Queues are replicated multiple times. If one queue fails, consumers of the queue may need to re-subscribe to the backup queue.

### 

### Additional optimization: