Saturday, April 10, 2021

System Design - How to design Google Drive / Dropbox (a cloud file storage service)?

29,808 views

System design is one of the most important aspect of software engineering. System design problems are usually open-ended discussions.

Let’s design a cloud file storage service like Google drive or drop box. These services provides a platform that allow users to upload their files online and access them from any device. In this article, we will focus on architecture design & performance analysis of the system.

So let’s dive right in!

Problem Statement

We need to design a service like Google Drive or Dropbox which allows users to store their data securely, synchronised & effectively on remote servers. User should be able to download and upload files from all their devices. System should be highly available, reliable and scalable.

Feature Expectations

The top level requirements of the system are as follows -

Functional Requirements -

- User should be able to download/upload, update, delete files from any device
- Files should be synchronised in all the devices that the user is logged in
- History/versioning of files (snapshotting of data)

Non-functional requirements

- Our service needs to be highly available
- The system should be highly reliable; any uploaded file should never be lost
- System should support large files uploading
- ACID operations :

Atomic - File upload should be all or none

Consistency - Both versions on device and server must be same

Isolation - Ensure multiple transactions at same time, with data consistency

Durability - Must be highly available & durable

Not in Scope

Sharing file access with other users, offline editing.

Scale

- **Total Users** : ~100 M

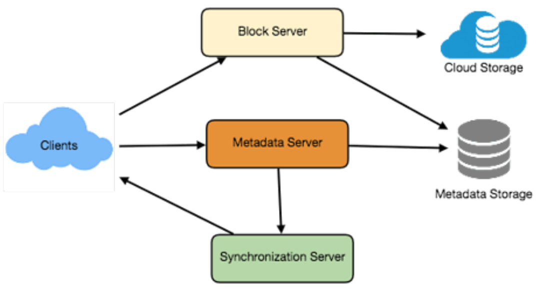
- **Daily Active Users** : ~50 M
- **QPS** : ~500M request per day (~9000 Queries Per Second)
- **Storage Estimate** : Assume, every user has avg 100 files (avg file size ~1MB) so we will have a total of 100B files. Hence, total storage would be -

  100MB x 100B = 10000PB.
- **Read/Write ratio** : ~ 1:1
- **Traffic Estimate** : ~5 GB new file writes per second
- **Memory Usage** : Assume, each user access 5 files daily and reading chunks of 200KB out of 1MB. So, following the 80-20 rule (80% traffic comes for 20% of the files), our cache size -

  ((50 M x 200KB x 5) x 20) / 100 = 10 Tera Byte
- **Active Connections** : 1 M active connections per minute

High Level Design

On a high level, we need to support two scenarios, one to upload/edit/delete files and other to view/search files. We would need some object storage server (cloud storage) to store files, one block server with whom client would interact, one database server to store metadata information (file name, file size, path etc) about the files and one synchronisation server.



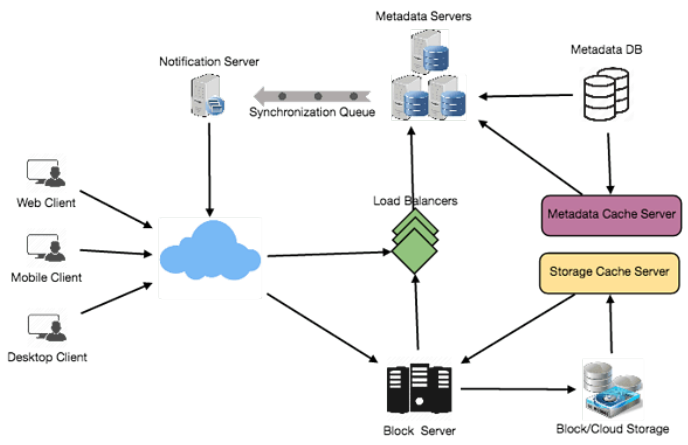
System APIs

Following are the APIs we can expose from our service -

1. *upload(string uploadToken, fileInfo file, userInfo user)*
2. *edit(string authToken, fileInfo file, userInfo user)*
3. *delete(string authToken, fileInfo file, userInfo user)*
4. *download(string authToken, fileInfo file, userInfo user)*

We can have an another API to generate a write token (uploadToken) before hand when a user request to upload a new file. Here we will have an opportunity to authenticate & validate the user. If we need to block a user from uploading new files, a check can be put here instead of allowing user to send whole file in request and then discard it. It can avoid unnecessary resources consumption.

Component Design & Deep dive



Let’s go through the major components of our system one by one:

Client

A client application on user's device talks to our service to upload, download and modify files to backend cloud storage. Below are some major operations done by client -

- **Upload/download file**

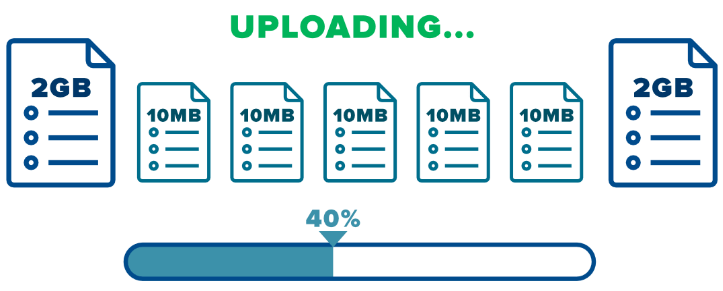
If we upload/download the file with full size, it will cost us storage and bandwidth. And also, latency will be increased for this process. If a user attempts to upload a 10MB file, that's a lot of resource usage :

1. 10MB of server memory tied up
2. A request handler being tied up for the entire amount of time to upload a 10MB file
3. High CPU Usage
4. Suppose, If uploading fails after 90%, the whole file would be required to upload again.
5. No efficient method to update the file for small edits as we would have the entire file, not chunks

It will hit scalability very hard. It's not a wise choice to use full file upload approach at this scale.

- **Handling file transfer efficiently**

We may divide the file into smaller chunks to make it easier to upload. Details of chunks can be included in metadata. Naming of chunks can be done by the hash value of chunks content.



This strategy will helps us in many ways -

1. **Bandwidth Optimisation** - Smaller chunks size optimises network bandwidth utilisation. While uploading, we can break the file into chunks and then upload it. In case of file upload failure, we don't need to upload whole file again, only failing chunk will be retried. If user updates the file, only modified chunk will be sent.

   Less amount of data transfer between client and server will reduce network bandwidth and most importantly, it will help us achieve a better response time.

2. **Cloud Storage utilisation** - As we are sending modified chunks only to the server in case of update instead of the entire file again, it will decrease the cloud storage

consumption.

3. **Latency or Concurrency Utilisation** - Transmitting the entire file at once consumes a lot more time as compared to small chunks. With multiple smaller data chunks, we can make use of concurrency also to upload/download the file using multi threading or multi processes.

4. **Faster lookup & version control** - As we are only transmitting the modified chunks in case of updates, it helps us in proving a history of versions of the file. We can directly lookup at the modified chunks to see the modifications.

- **How to calculate chunk size**

Calculation of optimal chunk size can be done based on below parameters -

1. Input/Output operations per second on cloud storage devices
2. Network bandwidth
3. Average file size in storage
4. Metadata information

- **How to implement synchronisation server**

We have a synchronisation server which ensures that any updates from one client as sync with other devices. But how can we implement client efficiently listening to changes happening with other clients?

Basic Solution

The very first basic solution would be, to periodically hit and check with the server for new changes. But this approach is highly inefficient as most of the time we would be getting empty responses. It is a waste of network bandwidth too and we are unnecessary putting extra load on the servers.

HTTP Long Polling

An effective solution would be to use HTTP long polling. With long polling, client does not expect immediate response from the server. Server keeps the request open and waits for the new changes. Whenever there are new modifications by any client, server would immediately send an HTTP response to the client.

NOTE - Unlike desktop or web clients, mobile clients usually sync on demand to save user's bandwidth and space.

Metadata database

We store the metadata & indexes of all chunks in our database as we have to track it. Important thing here is that we are not storing the actual file/chunks itself here, we are storing only the metadata information to retrieve the file later. It is also responsible for maintaining the history (versioning) of files.

Database can be a relational database such as MySQL or a non-relational database such as cassandra, dynamoDB or MongoDB. Let's discuss which database to consider for our use case -

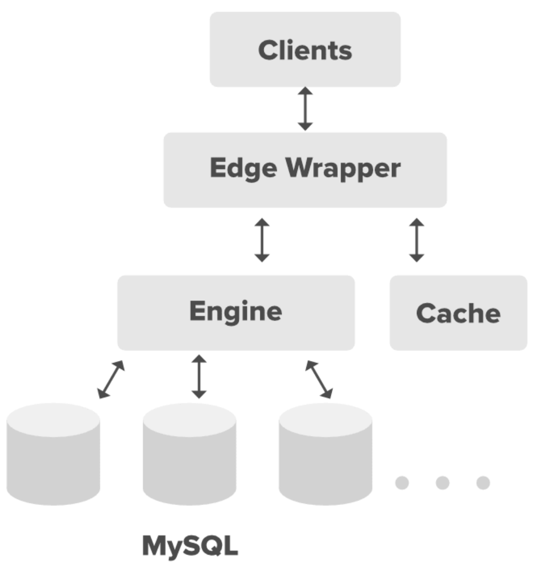
- **Relational vs Non-relational Database**

1. As, multiple users might be on the same file simultaneously & we need to ensure data consistency. Since NoSQL data storage do not support ACID properties in favour of performance and scalability. We would need to implement ACID properties pro-grammatically in synchronisation service. It would require extra DB configurations. (For example, Cassandra replication factor gives the consistency level). Here, relational database (MySQL) has a plus point as they support ACID properties.

2. Relational databases are difficult to scale as if we are using MySQL, we would need to use sharding or master slave techniques and it would become more difficult these multiple databases for any new updates.

To overcome this problem, we can have a wrapper around all databases and cache instead of directly talking to the database.

Metadata



NOTE : Storing file URL in database is a really bad idea. Using stored metadata information to construct the URL is a lot more robust and scalable.

Synchronisation Service

For every new update, synchronisation service is responsible to efficiently process updates and apply changes to other subscribed devices to keep their local db and remote db in sync.

Synchronisation service should be designed to transmit as less data as possible to avoid unnecessary network bandwidth and achieve a better response time. Server and client can calculate a hash (SHA-256) to check if chunk is updated or not. On server also, if we have the chunk with similar hash(even from another user), we can use the same chunk instead of creating a new copy to avoid data deduplication.

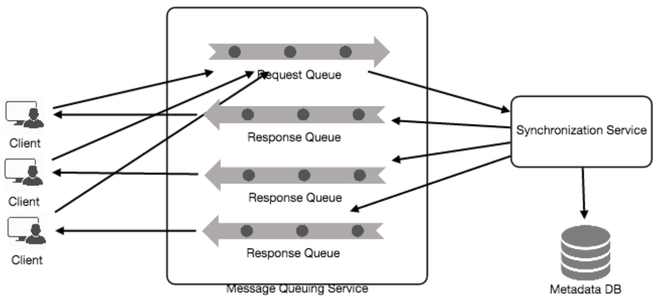
For more efficiency, we can consider using a communication middleware between client and sync service. It will provide a global message queue to support a very high number of pull or push from client.

Message Queuing Service

To enable asynchronous message based communication between client and server to serve huge number of requests, we can make use of a scalable message queuing service. It should support efficient storing of any number of messages in a highly available, reliable and scalable queue.

It will implement request and response queue. Request queue is a global queue which all client share. For each modification, the request will go to the global queue first to update the meta data DB and after that, synchronisation Service will update the meta data and update message is sent to all subscribed clients via response queue.

Along with high scalabiliy and high performance, it provides load balancing and elasticity for multiple instances of the Synchronisation Service. RabbitMQ, Apache Kafka, etc. are some of the examples of the messaging queue.



Cloud/Block Storage

Cloud/Block storage stores all the chunks, uploaded by users to the service. As we have metadata database, separate from storage, we can use any cloud storage approach eg - Amazon S3, Azure etc.

> *“When Dropbox started, they used S3 as block storage. However as they grew, they developed an in-house multi-exabyte storage system known as Magic Pocket. In magic Pocket, files are split up into blocks, replicated for durability, and distributed across data centres in multiple geographic regions.”*

Database Partitioning

From database scalability point of view, database partitioning comes into picture to ensure scalability. Partitioning is the database process where very large tables are divided into multiple smaller parts for faster queries.

- **Metadata Partitions**

1. **Based on first latter of file** :

   We can store file/chunks in partitions based on the first letter of a file path, means all files starting with same letter will stay in one partition. This is called "range based partitioning".

   The major disadvantage of this approach is that, it can lead to unbalanced servers. It might possible that some database partitions get filled completely as there are more files starting with same letter and some remains with no traffic.

2. **Based on hash of file ID** :

   Based on hash of fileId, randomly generated by our hash function, we can have partitions. Hash will be mapped to a number and it would be our partition to store the object. This approach can still lead to unbalanced partitions but can be solved by using consistent hashing.

Caching

Caching is a very common technique for performance. This is very helpful to lower the latency. For our use case, **Memcached** would be a good choice which can store the whole chunk, with it's respective hash. A cache for Metadata Database can also be used.

Cache servers are determined based on users' usage pattern & **LRU** (Least Recently Used) can be the optimal policy for our cache.

Load Balancer

We can adopt round robin or some other fancy algorithm for load balancing layer to distribute the incomming traffic uniformly.

> *"Source : INTERNET"*

Tweet this article