Designing Dropbox

Difficulty Level: Medium

Let's design a file hosting service like Dropbox or Google Drive.

Cloud file storage enables users to store their data on remote servers.

Usually, these servers are maintained by cloud storage providers and made available to users over a network (typically through the Internet).

Users pay for their cloud data storage on a monthly basis.

1. Why Cloud Storage?

Cloud file storage services have become very popular recently, as they simplify the storage and exchange of digital resources among multiple devices.

The shift from using single personal computers to using multiple devices with different platforms and operating systems, such as smartphones and tablets each with portable access from various geographical locations at any time, is believed to be accountable for the huge popularity of cloud services.

The following at the top benefits of such services

- **Availability**
  The motto of cloud storage services is to have data availability anywhere, anytime. Users can access their files/photos from any device whenever and wherever they like.

- **Reliability and Durability**
  Another benefit of cloud storage is that it offers 100% reliability and durability of the data.

  Cloud storage ensures that users will never lose their data by keeping multiple copies of the data stored on different geographically located servers.

- **Scalability**
  Users will never have to worry about getting out of storage space. With cloud

storage, you have unlimited storage as long as you are ready to pay for it.

2. Requirements and Goals of the System

What do we wish to achieve from a Cloud Storage system?

Here are the top-level requirements for our system:

1. Users should be able to upload and download their files/photos from any device

2. Users should be able to share files or folders with other users

3. Our service should support automatic synchronisation between devices, i.e. after updating a file on one device, it should get synchronised on all devices

4. The system should support storing large files up to a GB

5. ACID-ity is required — the atomicity, consistency, isolation and durability of all file operations should be guaranteed

6. Our system should support offline editing — users should be able to add, delete, and modify files while offline; as soon as they come online, all their changes should be synced to the remote server and other online devices

Extended Requirements

- The system should support snapshotting of the data, so that users can go back to any version of the files

3. Some Design Considerations

- We should expect huge read and write volumes

- Read to write ratio is expected to be nearly the same

- Internally, files can be stored in small parts or chunks (say 4MB); this can provide a lot of benefits, i.e. all failed operations shall only be retried for smaller parts of a file.

  If a user fails to upload a file, then only the failing chunk will be retried.

- We can reduce the amount of data exchange by transferring updated chunks only

- By removing duplicate chunks, we can save storage space and bandwidth usage

- Keeping a local copy of the metadata (file name, size etc.) with the clients, we can save a lot of round trips to the server

- For small changes, clients can intelligently upload the diffs instead of the whole chunk

4. Capacity Estimation and Constraints

Suppose that we have 500 million total users, and 100 million daily active users (DAU).

Let's assume that on average, each user connects from 3 different devices.

On average, if a user has 200 files, we will have 100 billion total files.

Assuming that our average file size is 100KB, this would lead us to need 10PB of total storage:

$$100 B \cdot 100 \, \text{KB} = 10 \, \text{PB}$$

Let's also assume that we will have one million active connections per minute.

5. High-Level Design

The user will specify a folder as the workspace on their device.

Any file or folder place in this folder will be uploaded to the cloud, and whenever a file is modified or deleted, it will be reflected in the same way on the cloud.

The user can specify similar workspaces on all their devices and any modification done on one device will be propagated to all other devices to have the same view of the workspace everywhere.

At a high-level, we need to store files and their metadata information like File Name, File Size, Directory etc., and who this file is shared with.

So, we need some servers that can help the clients to upload and download files to the cloud, and some servers that can facilitate updating metadata about files and users.

We also need some mechanism to notify all clients whenever an update happens so they can synchronise their files.

As shown in the diagram below:

- <u>Block Servers</u> will work with their clients to upload and download from cloud storage

- <u>Metadata Servers</u> will keep metadata of files updated in a SQL or NoSQL database

- <u>Synchronisation Servers</u> will handle the workflow of notifying all clients about different changes for synchronisation

High level design for Dropbox
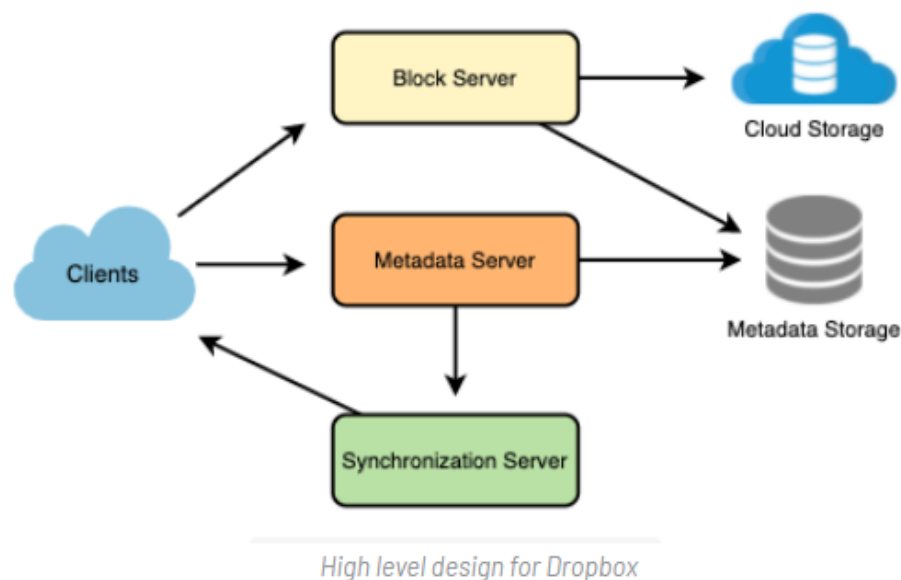
6. Component Design

Let's go through the major components of our system one-by-one.

a. Client

The client application monitors the workspace folder on the user's machine and syncs all files and folders in it with the remote cloud storage.

The client application will work with the storage servers to upload, download, and modify actual files to backend cloud storage.

The client interacts with the remote synchronisation service to handle any file metadata updates, e.g. change in the filename, file size, modification date etc.

Here are some of the essential operations for the client:

1. Upload and download files

2. Detect file changes in the workspace folder

3. Handle conflict due to offline or concurrent updates

How do we handle file transfer efficiently?

As mentioned above, we can break each file into smaller chunks so that we transfer only those chunks that are modified and not the whole file.

Let's say that we divide each file into fixed sizes of 4MB chunks.

We can statically calculate what could be an optimal chunk size based on:

1. Storage devices we use in the cloud to optimise space utilisation and input/output operations per second (IOPS)

2. Network bandwidth

3. Average file size in the storage etc.

In our metadata, we should also keep a record of each file and the chunks that constitute it.

Should we keep a copy of metadata with Clients?

Keeping a local copy of metadata not only enables us to do offline updates, but also saves a lot of round trips to update remote metadata.

How can clients efficiently listen to change happening with other clients?

One solution could be that the clients periodically check with the server if there are any changes.

The problem with this approach is that we will have a delay in reflecting changes locally as clients will be checking for changes periodically compared to a server notifying whenever there is some change.

If the client frequently checks the server for changes, it will not only be wasting bandwidth, as the server has to return an empty response most of the time, but will also be keeping the server busy.

Pulling information in this manner is not scalable.

A solution to the above problem could be to use <u>HTTP Long Polling</u>.

With long polling, the client requests information from the server with the expectation that the server may not respond immediately.

If the server has no new data for the client when the poll is received, instead of sending an empty response, the server holds the request open and waits for response information to become available.

Once it does have new information, the server immediately sends an HTTP/S response to the client, completing the open HTTP/S request.

Upon receipt of the server response, the client can immediately issue another server request for future updates.

Based on the above considerations, we can divide our client into four parts:

1. **Internal Metadata Database** will keep track of all the files, chunks, their versions, and their location in the file system

2. **Chunker** will split the files into smaller pieces called chunks.

   It will also be responsible for reconstructing a file from its chunks.

   Our chunking algorithm will detect the parts of the files that have been modified by the user, and only transfer those parts to the cloud storage; this will save us bandwidth and synchronisation time.

3. **Watcher** will monitor the local workspace folders and notify the Indexer of any action performed by the users, e.g. when users create, delete or update files and folders.

   The Watcher will also listen to any changes happening on other clients that are broadcasted by the synchronisation service.

4. **Indexer** will process the events received from the Watcher, and update the Internal Metadata Database with information about the chunks of the modified files.

   Once the chunks are successfully submitted/downloaded to the cloud storage,

the Indexer will communicate with the remote synchronisation service to broadcast changes to other clients and update the remote metadata database.
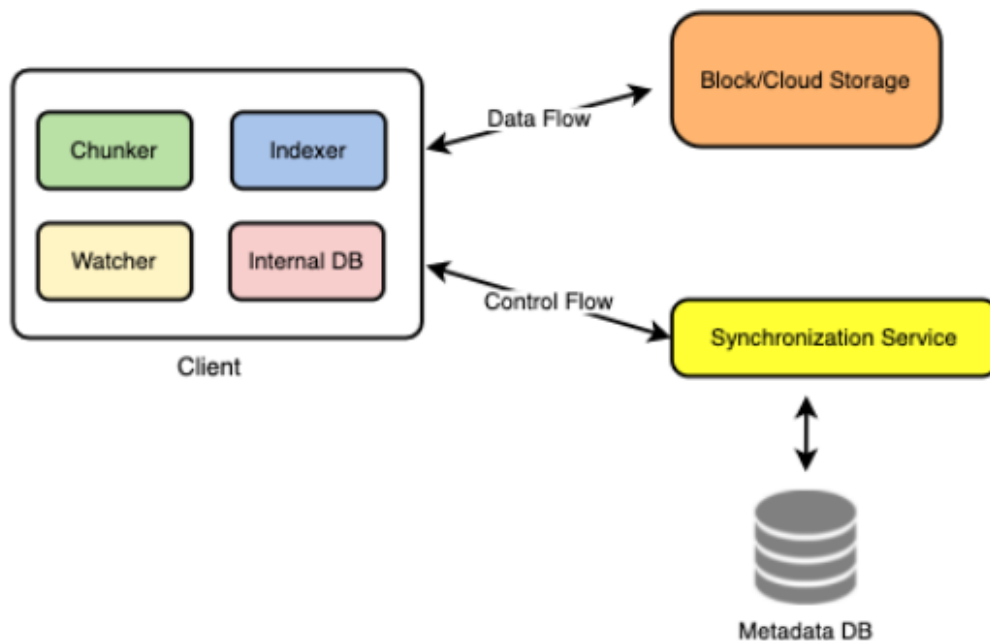
How should clients handle slow servers?

Clients should exponentially back-off if the server is busy/not responding. Meaning that if a server is too slow to respond, clients should delay their retries, and this delay should increase exponentially.

Should mobile clients sync remote changes immediately?

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

b. Metadata Database

The Metadata Database is responsible for maintaining the versioning and metadata information about files/chunks, users, and workspaces.

The Metadata Database can be a relational database such as MySQL, or a NoSQL database service like DynamoDB.

Regardless of the type of the database, the synchronisation service should be able to provide a consistent view of the files using a database, especially if more than one user is working with the same file simultaneously.

Since NoSQL data stores do not support ACID properties in favour of scalability and performance, we need to incorporate the support for ACID properties programmatically in the logic of our synchronisation service, in case we opt for this kind of database.

However, using a relational database can simplify the implementation of the synchronisation service, as they natively support ACID properties.

The metadata database should be storing information about the following objects:

1. Chunks
2. Files
3. User
4. Devices
5. Workspace (sync folders)

c. Synchronisation Service

The Synchronisation Service is the component that processes file updates made by a client, and applies these changes to other subscribed clients.

It also synchronises clients' local databases with the information stored in the remote Metadata Database.

The Synchronisation Service is the most important part of the system architecture due to its critical role in managing the metadata and synchronising users' files.

Desktop clients communicate with the Synchronisation Service to either obtain updates from the cloud, or send files and updates to the cloud, and, potentially, other users.

If a client was offline for a period of time, it will poll the system for new updates as soon as they come online.

When the Synchronisation Service receives an update request, it checks with the Metadata Database for consistency, and then proceeds with the update.

Subsequently, a notification is sent to all subscribed users or devices to report the file update.

The Synchronisation Service should be designed to transmit less data between clients and the cloud to achieve a better response time.

To meet this design goal, the Synchronisation Service can employ a differencing algorithm to reduce the amount of data that needs to be synchronised.

Instead of transmitting entire files from clients to the server or vice versa, we can just transmit the difference between two versions of a file.

Therefore, only the part of the file that has been changed is transmitted.

This also decreases bandwidth consumption and cloud data storage for the end-user.

As described above, we will be dividing our files into 4MB chunks and will be transferring modified chunks only.

Server and clients can calculate a hash (e.g. SHA-256) to see whether to update the local copy of a chunk or not.

On the server, if we already have a chunk with a similar hash (even from another user), we don't need to create another copy; we can just use the same chunk

To be able to provide an efficient and scalable synchronisation protocol, we can consider using a communication middleware between clients and the Synchronisation Service.

The messaging middleware should provide scalable message queueing and change notifications to support a high number of clients using pull or push strategies.

This way, multiple Synchronisation Service instances can receive requests from a global request queue, and the communication middleware will be able to balance its load.

d. Message Queueing Service

An important part of our architecture is a messaging middleware that should be able to handle a substantial number of requests.

A scalable Message Queueing Service that supports asynchronous message-based communication between clients and the Synchronisation Service best fits the requirements of our application.

The Message Queuing Service supports asynchronous and loosely couple message-based communication between distributed components of the system.

The Message Queueing Service should be able to efficiently store any number of messages in a highly available, reliable, and scalable queue.

The Message Queuing Service will implement two types of queues in our system.

The Request Queue is a global queue, and all clients will share it.

- Clients' requests to update the Metadata Database will be sent to the Request Queue first
- From there, the Synchronisation Service will take it to update metadata

The Response Queues that correspond to each individual subscribed client are responsible for delivering the update messages to each client.

- Since a message will be deleted from the queue once received by a client, we need to create separate Response Queues for each subscribed client to share update messages
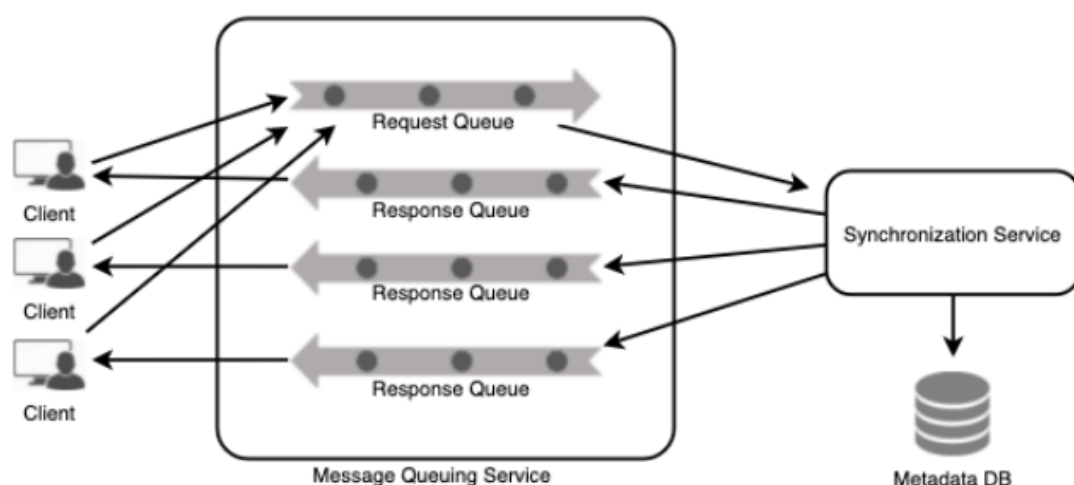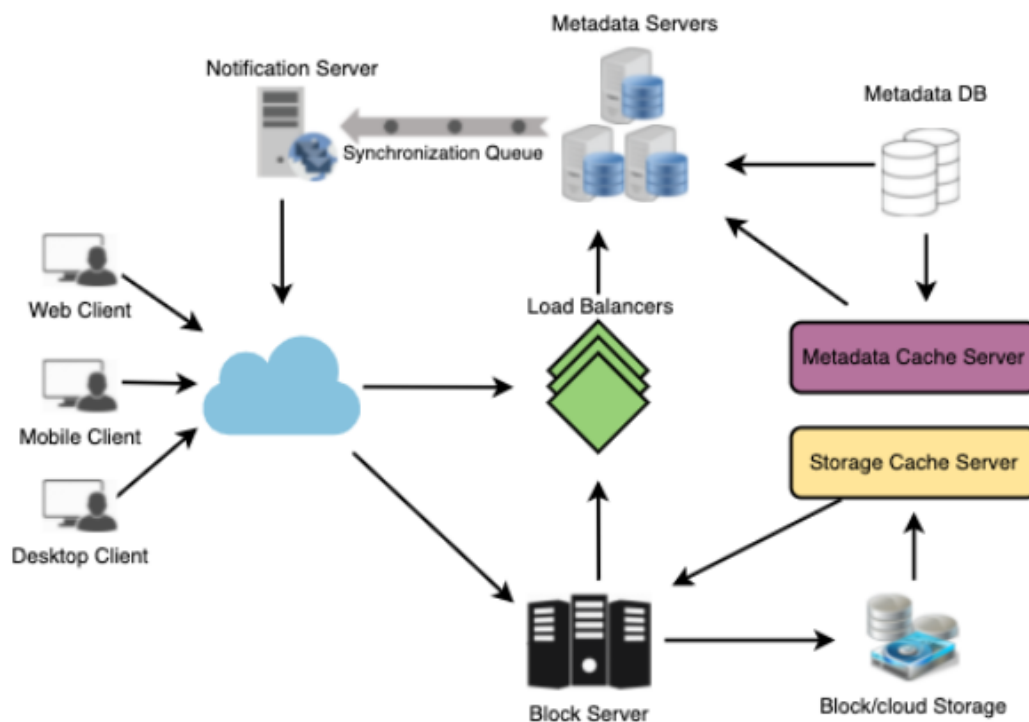
e. Cloud/Block Storage

Cloud/Block Storage stores chunks of files uploaded by the users.

Clients directly interact with the storage to send and receive objects from it.

Separation of the metadata from storage enables us to use any storage either in the cloud or in-house.

Detailed component design for Dropbox

7. File Processing Workflow

The sequence below shows the interaction between the components of the application in a scenario when Client A updates a file that is shared with Client B and C, so they should receive the update too.

If the other clients are not online at the time of the update, the Message Queueing Service keeps the update notifications in separate response queues for them until they come online later.

1. Client A uploads chunks to cloud storage

2. Client A updates metadata and commits changes

3. Client A gets confirmation, and notifications are sent to Clients B and C about the changes

4. Clients B and C receive metadata changes and download updated chunks

8. Data Deduplication

> **!** **Data Deduplication** is a technique used for eliminating duplicate copies of data to improve storage utilisation.

It can also be applied to network data transfer to reduce the number of bytes that must be sent.

For each new incoming chunk, we can calculate a hash of it, and compare that hash with all the hashes of the existing chunks to see if we we already have the same chunk present in our storage.

We can implement deduplication in two ways in our system, as seen below.

a. Post-Process Deduplication

With post-process deduplication, new chunks are first stored on the storage device and later some process analyses the data looking for duplication.

The benefit is that clients will not need to wait for the hash calculation or lookup to complete before storing the data, thereby ensuring that there is no degradation in storage performance.

The drawbacks of this approach, however, are:

1. We will unnecessarily be storing duplicate data — though for a short time

2. Duplicate data will be transferred consuming bandwidth

b. In-line Deduplication

Alternatively, deduplication hash calculations can be done in real-time, as the clients are entering data on their device.

If our system identifies a chunk that it has already stored, only a reference to the existing chunk will be added in the metadata, rather than a full copy of the chunk.

This approach will give us optimal network and storage usage.

9. Metadata Partitioning

To scale out the Metadata DB, we need to partition it so that it can store information about millions of users, and billions of files/chunks.

We need to come up with a partitioning scheme that would divide and store our data in different DB servers.

1. Vertical Partitioning

We can partition our database in such a way that we store tables related to one particular feature on one server.

For example, we can store all the user-related tables in one database, and all file/chunk-related tables in another database.

Although this approach is straightforward to implement, it has some issues:

1. Will we still having scaling issues? What if we have trillions of chunks to be stored, and our database cannot support storing such a huge number of records? How would we further partition such tables?

2. Joining two tables in two separate databases can cause performance and consistency issues. How frequently do we have to join user and file tables?

2. Range-based Partitioning

What if we store files/chunks in separate partitions based on the first letter of the file path?

In this case, we save all the files starting with the letter A in one partition, and those that start with the letter B into another partition and so on.

This approach is called range-based partitioning.

We can even combine certain less frequently occurring letters into one database partition.

We should come up with this partitioning scheme statically so that we can always store/find a file in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers.

For example, we decide to put all files starting with the letter E into a DB partition, and later we realise that we have too many files that start with the letter E, to the extent that we cannot fit them into one DB partition.

3. Hash-based Partitioning

In this scheme, we take a hash of the object we are storing, and based on this hash we figure out the DB partition to which this object should go.

In our case, we can take the hash of the FileID of the file object that we are storing to determine the partition in which the file will be stored.

Our hashing function will randomly distribute objects into different partitions, e.g. our hashing function can always map any ID to a number between [1…256], and this number would be the partition we will store our object in.

This approach can still lead to overloaded partitions, which can be solved by using Consistent Hashing.

10. Caching

We can have two kinds of caches in our system.

To deal with hot files/chunks we can introduce a cache for block storage. We can use an off-the-shelf solution like Memcached that can store whole chunks with their respective IDs/Hashes.

Then, before hitting Block storage, Block servers can quickly check for a cache hit.

Based on clients' usage patterns, we can determine how many cache server we need.

We will have a similar cache for Metadata DB.

Which cache replacement policy would best fit our needs?

When the cache is full, and we want to replace a chunk with a newer/hotter chunk, how would we choose which chunk to evict?

Least Recently Used (LRU) can be a reasonable policy for our system.

11. Load Balancer (LB)

We can add the load balancing layer at two place in our system:

1. Between clients and block servers

2. Between clients and metadata servers

Initially, a simple round robin approach can be adopted that distributes incoming requests equally among backend servers.

This LB is simple to implement, and does not introduce any overhead.

Another benefit of this approach is that if a server is dead, the LB will take it out of the rotation and will stop sending any traffic to it.

A problem with the round robin LB is that it won't take server load into consideration.

If a server is overloaded or slow, the LB will not stop sending new requests to that server.

To handle this, a more intelligent LB solution can be implemented that periodically queries backend servers about their load and adjusts traffic based on that.

12. Security, Permissions and File Sharing

One of the primary concerns that users will have while storing their files in the cloud is the privacy and security of their data, especially since in our system users can share their files with other users, or even make them public to share with everyone.

To handle this, we will store the permissions of each file in our metadata DB to reflect what files are visible or modifiable by any user — perhaps a list of UserIDs.