Designing Dropbox

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.

Similar Services: Google Drive
Difficulty Level: Medium

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 and their portable access from various geographical locations at any time is plausible.

LEARN TEACH

Availability: The motto of cloud storage services is to have data available 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 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.

If you haven't used Dropbox before, we would highly recommend creating an account there and uploading/editing a file and also going through different options their service offers. This will help you a lot in understanding this chapter better.

2. Requirements and Goals of the System

You should always clarify requirements at the beginning of the interview and should ask questions to find the exact scope of the system that interviewer is expecting you to design.

LEARN TEACH

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 synchronization between devices, i.e., after updating a file on one device, it should get updated on all other devices.

4. The system should support storing large files up to a GB.
5. ACID is very important in our scenario. 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/modify files while offline, and as soon as they come online, all their changes should be spread to the remote servers 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 requests.
- 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 e.g. all failed

LEARN TEACH

- 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 client can save us 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

- Let's assume that we have 500M total users, and 100M daily active users (DAU).
- Let's assume that on average each user connects from three different devices.
- On average if a user has 200 files/photos, we will have 100 billion total files.
- Let's assume that average file size is 100KB, this would give us ten petabytes of total storage.

$$100B * 100KB \rightarrow 10PB$$

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

LEARN TEACH

Total Users	500	million users

LEARN TEACH

File size (avg)	100	KB
Total Files	100	billion
Total Storage	10	PB

LEARN TEACH

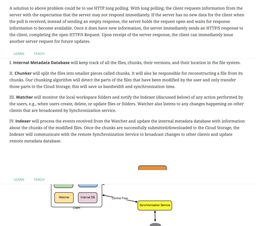
5. High Level Design

The user will specify a folder as the workspace on their device. Any files/photos/data placed 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 in the cloud storage. 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. We need a client and a server.

LEARN TEACH

As shown in the diagram below, Block servers will work with the clients to upload/download files from cloud storage, and Metadata servers will keep metadata of files uploaded in a SQL or NoSQL database. Synchronization servers will handle the workflow of notifying all clients about different changes for synchronization.



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 user's machine and syncs all files/folders it is 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 also interacts with the remote Synchronization Service to handle any file metadata updates e.g. changes to the file name, size, modification date, etc.

Here are some of the essential operations of the client:

1. Upload and download files.
2. Detect file changes in the workspace folder.
3. Handle conflicts 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 see we divide each file into fixed size of 4MB

LEARN TEACH

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

Should we keep a copy of metadata with Client? Keeping a local copy of metadata not only enable us to do offline updates but also saves a lot of round trips to update remote metadata.

How can clients efficiently listen to changes happening on 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 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 empty response most of the time but will also be keeping the server busy. Pulling information in this manner is not scalable too.

A solution to above problem could be to use HTTP long polling. 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 HTTPS response to the client, completing the open HTTPS Request. Upon receipt of the server response, the client can immediately issue another server request for future updates.

LEARN TEACH

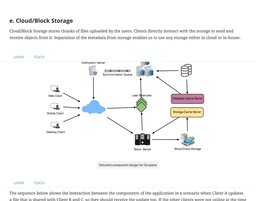
1. **Internal Metadata Database** will keep track of all the files, chunks, their versions, and their location in the file system.

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 synchronization time.

3. **Watcher** will monitor the local workspace folders and notify the Indexer (discussed below) of any action performed by the users, e.g., when users create, delete, or update files or folders. Watcher also listens to any changes happening on other clients that are broadcasted by the Synchronization 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 Synchronization Service to broadcast changes to other clients and update remote metadata database.

LEARN TEACH



How should clients handle slow servers? Clients should exponentially back off if the server is busy/not responding. Meaning, 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, that check for file changes on a regular basis, mobile clients usually sync on demand to save user's bandwidth and space.

b. Metadata Database

LEARN TEACH

Dynamo/Mr. Regardless of the type of the database, the Synchronization Service should be able to provide a consistent view of the files using a database especially if more than one user work with the same file simultaneously. Since NoSQL data stores do not support ACID properties in favor of scalability and performance, we need to incorporate the support for ACID properties programmatically in the logic of our Synchronization Service in case we opt for this kind of database. However, using a relational database can simplify the implementation of the Synchronization Service as they natively support ACID properties.

Metadata Database should be storing information about following objects:

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

LEARN TEACH

the metadata and synchronizing users' files. Desktop clients communicate with the Synchronization Service to either stream updates from the Cloud Storage or send files and updates to the Cloud Storage and potentially other users. If a client was offline for a period, it polls the system for new updates as soon as it becomes online. When the Synchronization Service receives an update request, it checks with the database for consistency and then the changes are proceeds with the update. Subsequently, a notification is sent to all subscribed users or devices to report the file update.

The Synchronization Service should be designed in such a way to transmit less data between clients and the Cloud Storage to achieve better response time. To meet this design goal, the Synchronization Service can employ a differencing algorithm to reduce the amount of the data that needs to be synchronized. 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 transferred. This has other benefits as well, for example it would cost less 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 maintain a hash (e.g., SHA-256) to see whether to update the local copy of a chunk or not. On 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 use the same chunk. This is discussed in detail later under Data Deduplication.

To be able to provide an efficient and scalable synchronization protocol we can consider using a communication middleware between clients and the Synchronization Service. The messaging middleware should provide scalable message

LEARN TEACH

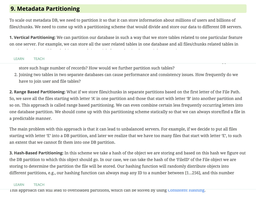
will be able to balance their load.

d. Message Queuing 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 Queuing Service that supports asynchronous message-based communication between clients and the Synchronization Service instances best fits the requirements of our application. The Message Queuing Service supports asynchronous and loosely coupled message-based communication between distributed components of the system. The Message Queuing Service should be able to efficiently store any number of messages in a highly available, reliable and scalable queue.

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 Synchronization Service will take it to update metadata. The Response Queues that correspond to individual subscribed clients 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.

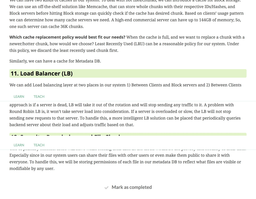
LEARN TEACH



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 cloud or in house.

LEARN TEACH



Detailed component design for Dropbox

LEARN TEACH

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 were not online at the time of the update, the Message Queuing Service keeps the update notifications in separate response queues for them until they become 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. Client 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 utilization. It can also be applied to network data transfers 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 already have some chunk present in our storage.

We can implement deduplication in two ways in our system:

LEARN TEACH

Write post-process deduplication, new chunks are first stored on the storage device, and later some process analyzes 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. Drawbacks of this approach are 1) We will unnecessarily be storing duplicate data, through 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 which it has already stored, only a reference to the existing chunk will be added in the metadata, rather than the full copy of the chunk. This approach will give us optimal network and storage usage.

9. Metadata Partitioning

To scale our 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 to different DB servers.

LEARN TEACH

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 files/chunks related tables in

LEARN TEACH

store such huge number of records? How would we further partition such tables?

1) Joining user 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/Chunk. So, we save all the files starting with letter 'A' in one partition and those that start with 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 storefind a file in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers. For example, if we decide to put all files starting with letter 'S' into a DB partition, and later we realize that we have too many files that start with letter 'S', to such an 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 we are storing to determine the partition for the file will be stored. Our hashing function will randomly distribute objects into different partitions, e.g., our hashing function can return a number between [1, 256], and this number

LEARN TEACH

can aggregate can also cause to overcrowded partitions, which can be worse

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 Memcache, that can store whole chunks with their respective IDs/hashes, and Block servers before hitting Block storage can quickly check if the cache has desired chunk. Based on clients' usage pattern we can determine how many cache servers we need. A high-end commercial server can have up to 144GB of memory. So, one such server can cache 36K chunks.

Which cache replacement policy would best fit our needs? When the cache is full, and we want to replace a chunk with a newer/fresher one, look-back LRU (Least Recently Used [LRU]) can be a reasonable policy for our system. Under this policy, we discard the least recently used chunk first.

Similarly, we can have a cache for Metadata DB.

11. Load Balancer (LB)

We can add Load balancing layer at two places in our system 1) Between Clients and Block servers and 2) Between Clients

LEARN TEACH

approach it is a server is dead, LB will take it out of the rotation and will stop sending any traffic to it. A problem with Round Robin LB is 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 placed that periodically queries backend server about their load and adjusts traffic based on that.

LEARN TEACH

Especially since in our system users can share their files with other users or even make them public to share it with everyone. To handle this, we will be storing permissions of each file in our metadata DB to reflect who files are visible or modifiable by any user.

Mark as completed

Designing Instagram

Next
Designing Facebook Messenger

Send feedback or ask a question

22 recommendations