Dropbox System Design

28 APR 2020 on System Design

Dropbox is a cloud storage service which allows users to store their data on remote servers. The remote servers store files durably and securely, and these files are accessible anywhere with an Internet connection.

In this design, we will focus mainly on availability, reliability and scalability. So let's dive right in!

Requirements

Let's look in to some of the functional and non-functional requirements before we start to design the system.

Functional Requirements

1. Users should be able to sign up using their email address and subscribe to a plan. If they don't subscribe, they will get 1 GB of free storage.
2. Users should be able to upload and download their files from any device.
3. Users should be able to share files and folders with other users.
4. Users should be able to upload files up to 1 GB.
5. System should support automatic synchronization across the devices.
6. System should support offline editing. Users should be able to add/delete/modify files offline and once they come online, changes should be synchronized to remote server and other online devices.

Non-functional requirements

1. System should be highly reliable. Any file uploaded should not be lost.
2. System should be highly available.

Out of scope

1. Real-time collaboration on a file.
2. Versioning of the file.

Capacity Estimation

Let's do some back-of-the-envelope calculations to estimate the bandwidth and storage required.

Assumptions

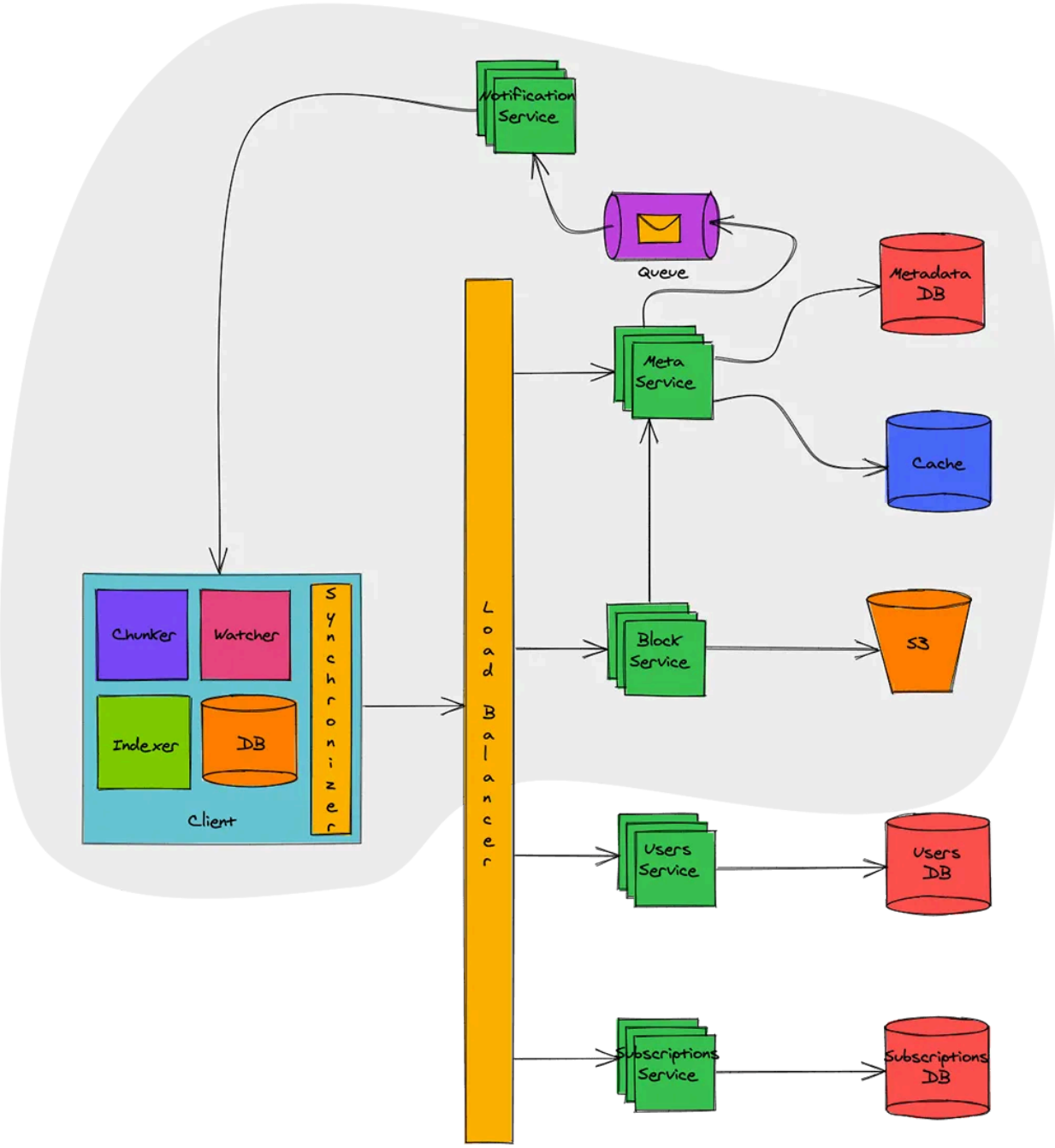
1. Total number of users = 500 million
2. Total number of daily active users = 100 million
3. Average number of files stored by each user = 200
4. Average size of each file = 100 KB
5. Total number of active connections per minute = 1 million

Storage Estimations

Total number of files = 500 million * 200 = 100 billion
Total storage required = 100 billion * 100 KB = 10 PB

Detailed Component Design

The system needs to deal with huge volume of read and write data and their ratio will remain almost same. Hence while designing the system, we should focus on optimizing the data exchange between client and server.



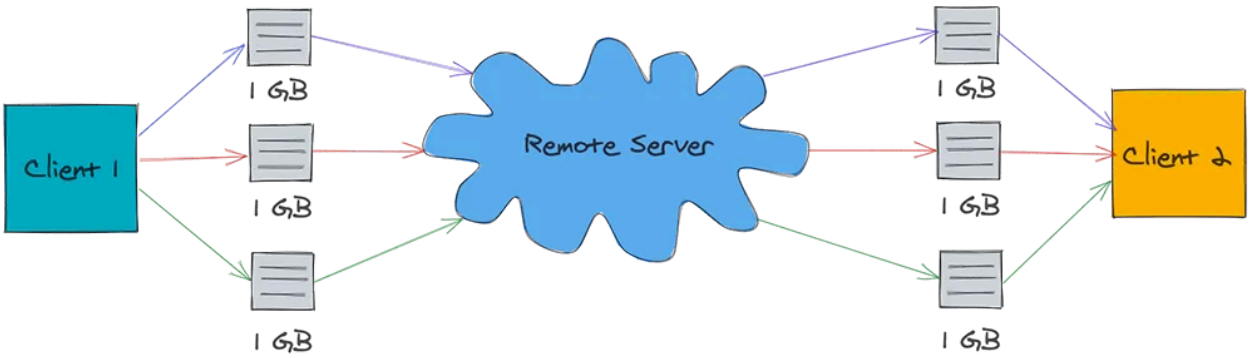
Our focus is on building the components which are in grey area in diagram above. Other components, which are outside, like Users Service and Subscriptions Service have been discussed in detail previously. Hence let's look in to remaining components in detail.

Client

Client here means desktop or mobile application which keeps a watch on user's workspace and synchronizes the files with remote server. Below are some main responsibilities of client:

- Watch workspace for changes
- Upload or download changes to file from remote server
- Handle the conflicts due to offline or concurrent updates
- Update file metadata on remote server if they change

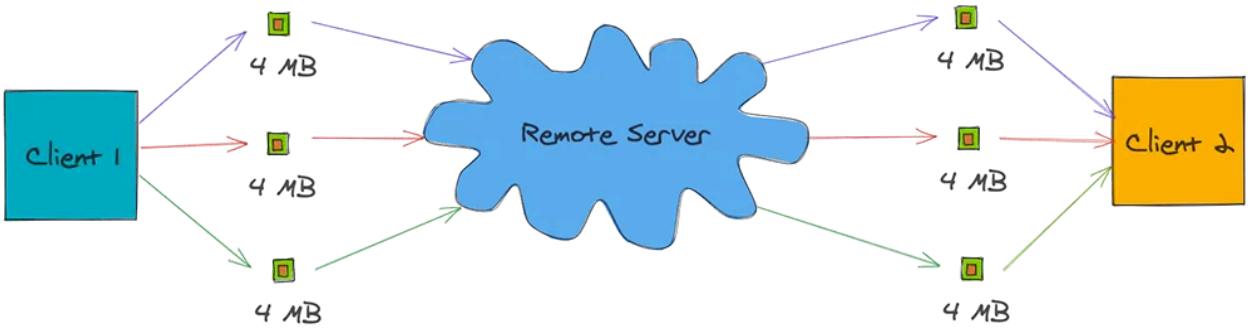
Let's naively assume that we build a client which synchronizes the file on each modification to remote server.



As shown in image above, let's assume that there is a file of 1 GB and three successive small changes were made to this file. Due to this, the file is sent three time to remote server and same is downloaded three times on another client. In this whole process, 3 GB of bandwidth was used for upload and 3 GB of bandwidth was used for download. Further, the download bandwidth increases in proportion of clients watching the file.

Did you observe that a total of 6 GB of bandwidth was used for just a small change? Also if there is a loss in connectivity, client has to upload/download entire file again. This is a huge waste of bandwidth and hence let's try to optimize it.

Now let's assume we build a client which breaks the file in to smaller chunks of say 4 MB and uploads them to remote server as shown below:



If there is any change in file, client determines which chunk has changed and just uploads that chunk to remote server. Similarly on other side, other client gets notified about the chunk that has changed and downloads just that chunk. This way just 24 MB of bandwidth was used, in contrast to 6 GB earlier, for synchronizing three small changes to the file.

Keeping this in mind, let's look in to the different components of this optimized client:

Client Metadata Database: Client Metadata Database stores the information about different files in workspace, file chunks, chunk version and file location in the file system. This can be implemented using a lightweight database like SQLite.

Chunker: Chunker splits the big files in to chunks of 4 MB. This also reconstructs the original file from chunks.

Watcher: Watcher monitors for file changes in workspace like update, create, delete of files and folders. Watcher notifies Indexer about the changes.

Indexer: Indexer listens for the events from watcher and updates the Client Metadata Database with information about the chunks of modified file. It also notifies Synchronizer after committing changes to Client Metadata Database.

Synchronizer: Synchronizer listens for events from Indexer and communicates with Meta Service and Block service for updating meta data and modified chunk of file on remote server respectively. It also listens for changes broadcasted by Notification Service and downloads the modified chunks from remote server.

Meta Service

Meta Service is responsible for synchronizing the file metadata from client to server. It's also responsible to figure out the change set for different clients and broadcast it to them using Notification Service.

When a client comes online, it pings Meta Service for an update. Meta Service determines the change set for that client by querying the Metadata DB and returns the change set.

If a client updates a file, Meta Service again determines the change set for other clients watching that file and broadcasts the change set via Notification Service.

Meta Service is backed by Metadata DB. This database contains the metadata of file like name, type (file or folder), sharing permissions, chunks information etc. This database should have strong ACID (atomicity, consistency, isolation, durability) properties. Hence a relational database, like MySQL or PostgreSQL, would be a good choice.

Since querying the database for every synchronization request is a costly operation, a in-memory cache is put in front of Metadata DB. Frequently queries data is cached in this cache thereby eliminating the need of database query. This cache can be implemented using Redis or Memcached and write-around cache strategy can be applied for optimal performance.

> *Dropbox uses a claver algorithm for efficiently synchronizing files across multiple clients. You can read more details about the same here.*

Block Service

Block Service interacts with block storage for uploading and downloading of files. Clients connect with Block Service to upload or download file chunks.

When a client finishes downloading file, Block Service notifies Meta Service to update the metadata. When a client uploads a file, Block Service on finishing the upload to block storage, notifies the Meta Service to update the metadata corresponding to this client and broadcast messages for other clients.

Block Storage can be implemented using a distributed file system like Glusterfs or Amazon S3. Distributed file system provides high reliability and durability

making sure the files uploaded are never lost.

> *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 centers in multiple geographic regions.*

Notification Service

Notification Service broadcasts the file changes to connected clients making sure any change to file is reflected all watching clients instantly.

Notification Service can be implemented using HTTP Long Polling, Websockets or Server Sent Events. Websockets establish a persistent duplex connection between client and server. It is not a good choice in this scenario as there is no need of two way communication. We only need to broadcast the message from service to client and hence it's an overkill.

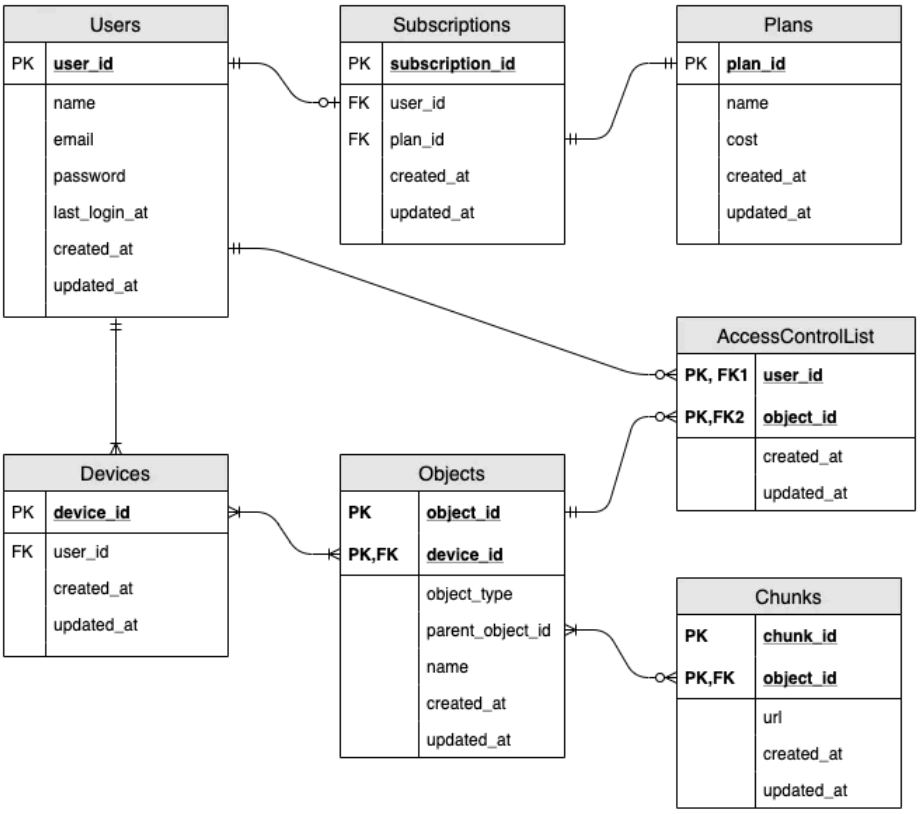
HTTP Long Polling is a better choice as server keeps the connection hanging till a data is available for client. Once data is available, server sends the data closing the connection. Once the connection is closed, client has to again establish a new connection. Generally, for each long poll request, there is a timeout associated with it and client must establish a new connection post timeout.

In Server Sent Events, client establishes a long term persistent connection with server. This connection is used to send events from server to client. There is no timeout and contain remains alive till the client remains on network. This fits our use case perfectly and would be a good choice for designing Notification Service. Though the Server Sent Events are not supported in all browsers, it's not a concern for us as we have our custom built desktop and mobile clients where we can utilize it.

Notification Service before sending the data to clients, reads the message from a message queue. This queue can be implemented using RabbitMQ, Apache ActiveMQ or Kafka. Message queue provides a asynchronous medium of communication between Meta Service and Notification Service and thus Meta Service need not to wait till notification is sent to clients. Notification service can keep on consuming the messages at it's own pace without affecting the performance Meta Service. This decoupling also allows us to scale both services independently.

Database Schema

The database schema containing most important tables is illustrated below:



Let's quickly sum up above database schema:

- A user may subscribe for paid service.
- Each subscription must have one plan.
- Each use must have at-least one device.
- Each device will have at-least one object (file or folder). Once user registers, we create a root folder for him/her making sure he/she has at-least one object.
- Each object may have chunks. Only files can have chunk, folders can't have chunks.
- Each object may be shared with one or multiple users. This mapping is maintained in AccessControlList.

APIs

The service will expose API for uploading file and downloading file. Other API like user sign-up, sign-in, sign-out, subscribing, unsubscribing etc. have already been discussed in this article.

Download Chunk

This API would be used to download the chunk of a file.

Request:

```
GET /api/v1/chunks/:chunk_id
X-API-Key: api_key
Authorization: auth_token
```

Response:

```
200 OK
Content-Disposition: attachment; filename="<chunk_id>"
Content-Length: 4096000
```

The response will contain `Content-Disposition` header as `attachment` which will instruct the client to download the chunk. Note that `Content-Length` is set as `4096000` as each chunk is of 4 MB.

Upload Chunk

This API would be used to upload the chunk of a file.

Request:

```
POST /api/v1/chunks/:chunk_id
X-API-Key: api_key
Authorization: auth_token
Content-Type: application/octet-stream
/path/to/chunk
```

`Content-Type` header is set as `application/octet-stream` to tell server that a binary date is being sent.

Response:

```
200 OK
```

On successful upload, the server will return HTTP response code `200`. Below are some possible failure response codes:

```
401 Unauthorized
400 Bad request
500 Internal server error
```

Get Objects

This API would be used by clients to query Meta Service for new files/folders when they come online.

Request:

```
GET /api/v1/objects?local_object_id=<Max object_id present
locally>&device_id=<Unique Device Id>
X-API-Key: api_key
Authorization: auth_token
```

Client will pass the maximum object id present locally and the unique device id.

Response:

```
200 OK
{
  new_objects: [
    {
      object_id:
      object_type:
      name:
      chunk_ids: [
        chunk1,
        chunk2,
        chunk3
      ]
    }
  ]
}
```

Meta Service will check the database and return an array of objects containing name of object, object id, object type and an array of `chunk_ids`. Client calls the Download Chunk API with these `chunk_ids` to download the chunks and reconstruct the file.

Performance

Chunking of files and data deduplication while uploading files boosts the performance a lot. Let's look in to both of these in detail.

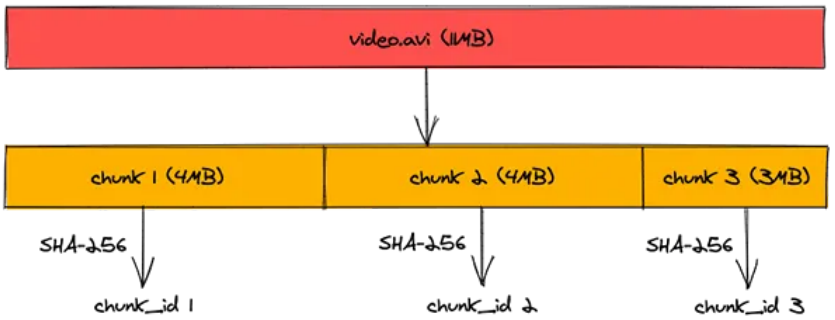
Chunking Files

Instead of uploading entire file in one go, we are chunking files in blocks of 4 MB and then uploading each chunk. This helps in improving performance in following ways:

- Multiple chunks can be uploaded in parallel thereby reducing the time for upload.
- If there is a small change in file, only the affected chunk is uploaded and downloaded by other clients, saving us bandwidth and time.
- If file upload is interrupted due to network connectivity loss, instead of uploading/downloading entire file again, we can just resume after the last chunk already uploaded/downloaded. This again saves our bandwidth and time.

Data Deduplication

The files are getting chunked in to blocks of 4 MB and each chunk is assigned a `chunk_id` which is SHA-256 hash of that chunk.



When a client tries to upload a chunk whose `chunk_id` is already present in Metadata DB, Meta Service just adds a row in `Chunks` table containing the new `object_id` and `chunk_id`. This eliminates the need to re-uploading the data thereby saving bandwidth and time. This also helps us in saving the space of block storage as there won't be multiple copies of one chunk in same data center.

Caching

We are using an in-memory caching to reduce the number of hits to Metadata DB. In-memory caches like Redis and Memcached cache the data from database in key-value pair.

From Meta Service servers, before hitting the Metadata DB, we are checking if data exists in cache or not. If it exists then we return the value from there itself bypassing a database trip. However if data is not present in cache, we hit the database, getting the data and populating the same in cache too. Hence subsequent requests won't hit the database and get the data from cache itself. This caching strategy is known as write-around caching.

Least Recently Used (LRU) eviction policy is used for caching data as it allows us to discard the keys which are least recently fetched.

Scalability

Our architecture is highly scalable owing to following facts:

Horizontal Scaling

We can add more servers behind the load balancer to increase the capacity of each service. This is known as Horizontal Scaling and each service can be independently scaled horizontally in our design.

Database Sharding

Metadata DB is sharded based on `object_id`. Our hash function will map each `object_id` to a random server where we can store the file/folder metadata. To query for a particular `object_id`, service can determine the database server using same hash function and query for data. This approach will distribute our database load to multiple servers making it scalable.

Cache Sharding

Similar to Metadata DB Sharding, we are distributing the cache to multiple servers. In-fact Redis has out of box support for partitioning the data across

multiple Redis instances. Usage of Consistent Hashing for distributing data across instances ensures that load is equally distributed if one instance goes away.

Security

Our system is highly secure due to following:

HTTPS

The traffic between client and server is encrypted over HTTPS. This ensures that no one in the middle is able to see the data, especially the file contents.

Authentication

For each API request post log-in, we are doing authentication by checking the validity of `auth_token` in `Authorization` HTTP header. This makes sure that requests which originate from clients are legitimate.

Resiliency

Our system highly resilient owing to following:

Distributed Block Storage

Files are split up in to chunks and these chunks are replicated within data center for durability. Also these chunks are distributed across data centers in multiple geographic regions for redundancy. This makes sure that enough copies of chunks are available within data center is one machine goes down. Also if entire data center goes down, chunks can be served from a data center in other geographical location.

Queuing

We are using queuing in our system for sending out the notification to clients. Hence if any worker dies, message in a queue isn't acknowledged and other worker picks up the task again.

Load Balancing

Since we are putting multiple servers behind the load balancer, there is redundancy. Load Balancer is continuously doing health check on servers behind it. If any server dies, load balancer stops forwarding the traffic to it and removes it from cluster. This makes sure that requests don't fail due to a unresponsive server.

Geo-redundancy

We are deploying exact replica of our services in data-centers across multiple geographical locations. This ensures that if one data-center goes down due to some reason, the traffic could still be served from remaining data-centers.

References

- Dropbox Architecture Overview
- How We've Scaled Dropbox

- Inside the Magic Pocket
- Streaming File Synchronization

Pranav

Hi, I'm Pranav. I love to learn and write about architectures of highly scalable & distributed systems.

Share this post

4 Comments

Login ▾

G

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

♡ 8 Share

Best Newest Oldest

Jack
3 years ago edited

NotificationService can contain multi instances working at the same time, hence a consumed message from NotificatinServiceA has the possibility that it doesn't maintain a connection with the client, that connection is handled by NotifcationServiceB, how can this be resolved?

o o Reply

K

K D
4 years ago

I think this is a very well written piece on this design.
One suggestion is that I feel that you should decouple the MetaDataService and SynchronisationService on the server end. SyncSevice has multiple responsibilities of its own like syncing with different devices and other users sharing the same file etc.

o o Reply

S

Sameer Srinivas
4 years ago

Have a question on Chunks. How would Chunker (at client side) know how to assemble chunks of a file? Shouldn't there be a field like chunk_no to identify the position of a chunk in the file? Thanks.

o o Reply

J

Jialong Wan ↗ Sameer Srinivas
4 years ago

I think so. Chunk table or metadata DB should contains chunk_order_num to help maintain orders of chunks, and will be used for aggragation.

o o Reply

Subscribe **Privacy** **Do Not Sell My Data**