Dropbox / Google Drive System Design

Date: 22nd Feb 2025
Time Taken: 90 Mins

Qs for the Interviewer

1. What all operation user can perform
2. total User? DAU? Avg file size? File type? Avg file per user?
3. Web application? Mobile application? Desktop application?

Requirements

Functional Requirements

1. User able to create a new file - it get synced with cloud and other clients
2. User able to update new file - it get synced with cloud and other clients
3. User able to see / access older version of file
4. Optional: User can share a file with other user, both user can edit, concurrent change / overwrite control

Non Functional Requirements

1. Low latency in sync of data between devices
2. High consistency for data between devices

Entity

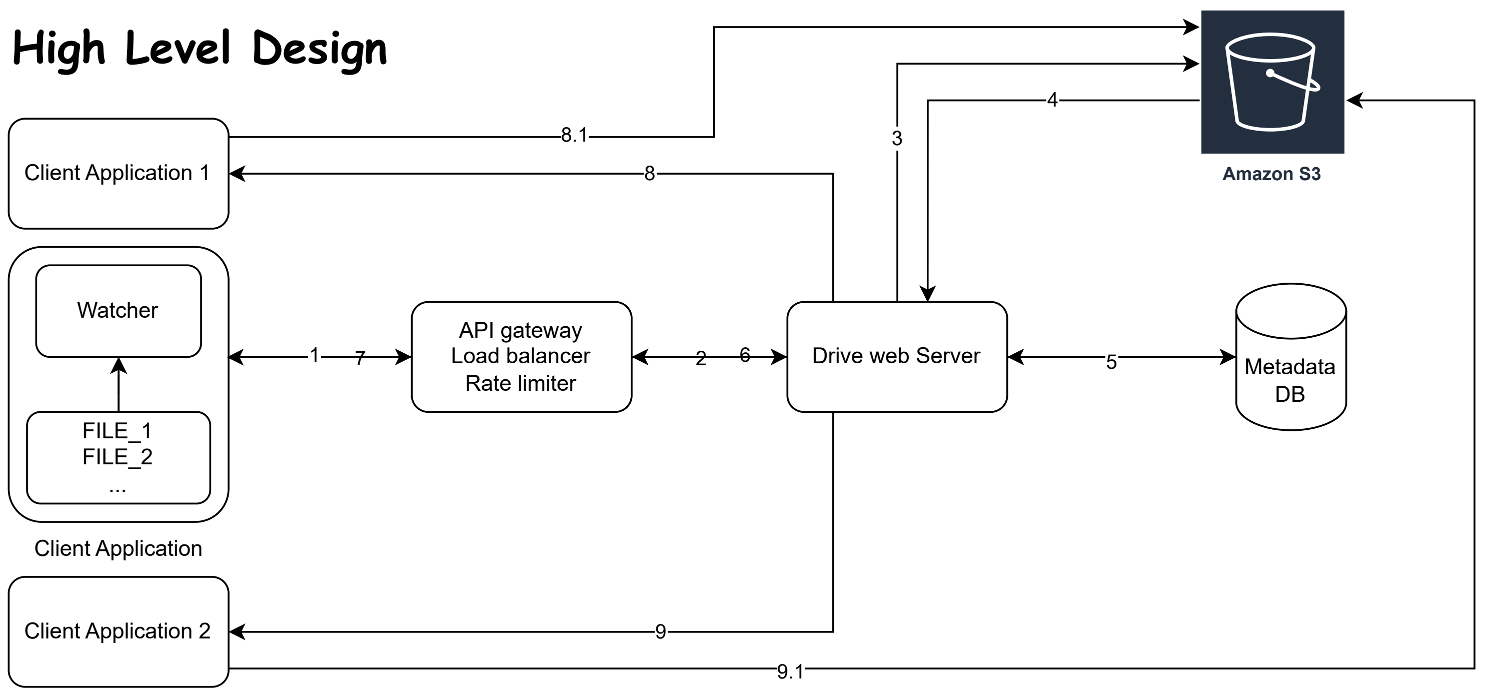
1. User
2. Client
3. File

APIs

1. User created a new file
POST /upload
```
body {
	file_path:"",
	file_name:"",
	file_size:"",
}
```

2. User updated a existing file
POST /upload
```
body {
	file_path:"",
	file_name:"",
	file_size:"",
}
```

High Level Design



Design Draft 1

Flow:

1. On client device we will have clinet application that is on watch, as a new file is created by user it is been uploaded to drive web server
2. File been uploaded to the web server
3. File is been updated to a blob storage like S3
4. S3 links for file storage returned
5. File and S3 metadata stored in DB
6. 7. File uploaded successfully response to clinet
8. 9. Other clients are sent with S3 link that download the file from s3

Drawbacks:

1. File been uploaded 2 times - once one web server and 2nd on s3 - wastage to bandwidth
2. If file is 50GB, and during upload any network issue happens - one need to re upload whole file again - bad customer experience.
3. Client application 1 and 2 downloads the complete file again and again on each edit - wastage of bandwidth
4. Other clients application may be offline - how to send them the update?

Deep Dive in Design

Fix: Multiple time file upload

Once a new file is created, we can just tell about file metadata to to drive web server that will create a bucket / space on S3 and provide the desktop / app client with the link path on which the file can be directly uploaded

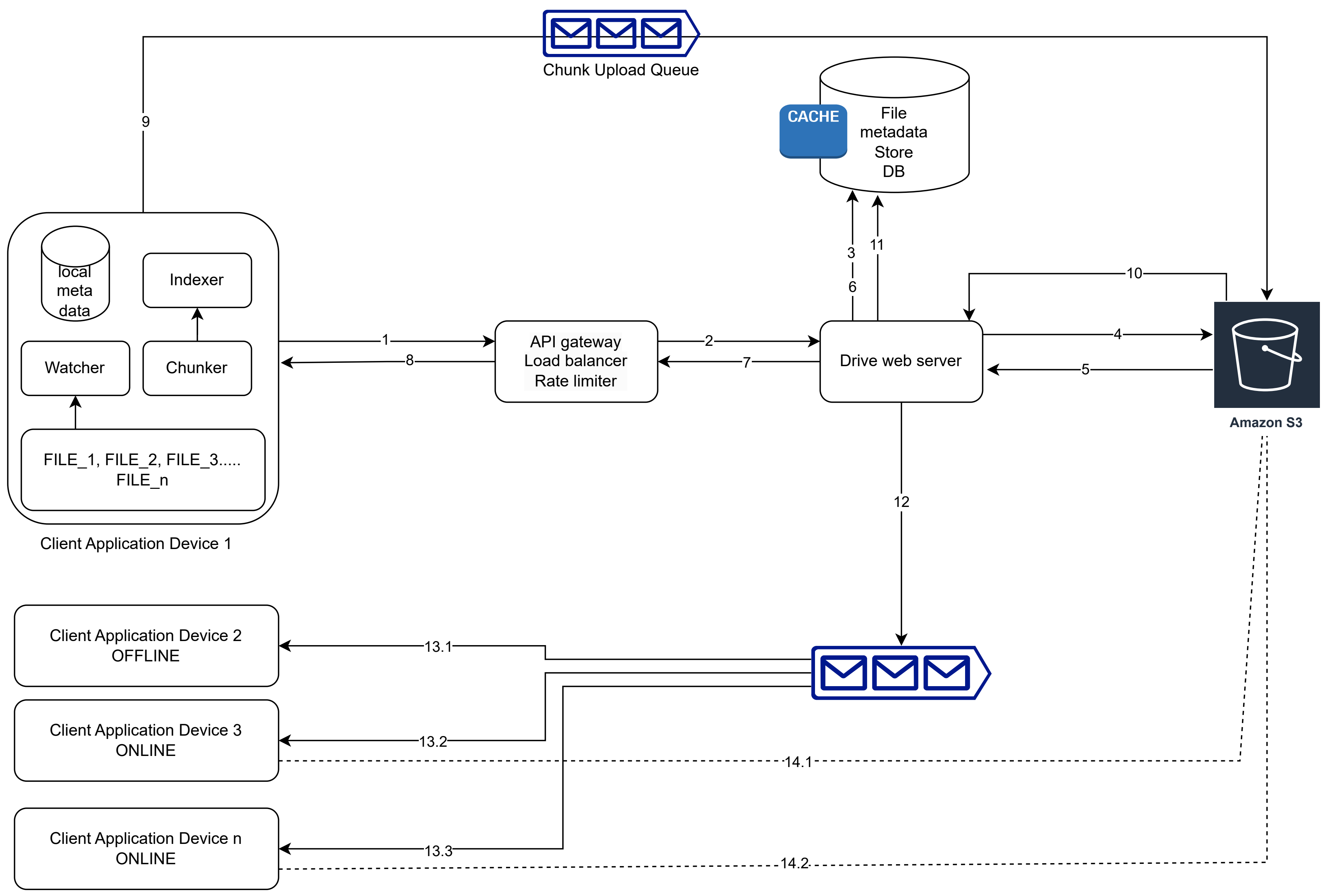
Fix: Upload large files & download by the other client application

In place of having a large file, we can break the file in smaller units called chunks, each chunk can be of size 4 MB, and can be uniquely identified using the hash of the chunk

Fix: Other clients application may be offline - how to send them the update

We can send the update notification to every client and each client can have a queue, as client comes online it can pick update and download from the queue, which missing any stuff.

Updated High Level Design



Chunk Upload Queue
9
CACHE
File metadata Store DB
3
6
11
10
local meta data
Indexer
Watcher
Chunker
FILE_1, FILE_2, FILE_3..... FILE_n
Client Application Device 1
1
8
API gateway Load balancer Rate limiter
2
7
Drive web server
4
5
Amazon S3
12
Client Application Device 2 OFFLINE
13.1
Client Application Device 3 ONLINE
13.2
14.1
Client Application Device n ONLINE
13.3
14.2

Flow

1. User created / Added a new file in the device
   - 1.1 Watcher got to know about new file - get its metadata (name, size etc)
   - 1.2 Chunker create smaller chunk of file of size ~ 4mb
   - 1.3 Indexer hashes the smaller chunk to identify each chunk
   - 1.4 Local metadata DB store all info about file metadata, chunks info
2. File metadata sent to API gateway that passes the metadata to drive web server
3. File metadata stored in the drive DB
4. Drive web server tells about the metadata of file to S3 and request it for a URL on which exact same file can be uploaded
5. S3 returns the URL
6. S3 URL details and path of file stored in Drive DB
7. 8. Drive web server returns the S3 link to user's client application
9. Client application star uploading the chunks to provided S3 link
   > a queue can be used here to increase reliability
10. As file chunks are getting uploaded to the S3, S3 tells about the chunk info and links for each chunk to the derive web server
11. Drive saves all the chunk info in the drive database
12. Once all the upload is completed, from DB we check other device info for same user and for each device we sent the info regarding life metadata, its chunk info and the se path
13. Whenever the client device comes online/connects to the internet it will pull the message from the queue
14. Each client device has the s3 link, it can download the relevant file chunks from the S3

Senario: An existing file is edited.
1. Only the file chunk that has been uploaded will be impacted
2. Only the file chunk that is impacted will be downloaded

Database Entity Design

User
1. UserID (PK) 2. Name 3. Email 4. List<Devices>

Devices
1. DeviceID (PK) 2. Name 3. UserID (FK) 4. CreatedOn 5. LastUpdatedOn

File
1. FileID (PK) 2. UserID (FK) 3. CreatedOn 4.LastUpdatedOn 5.List<ChunkID> 3. S3Link

Chunk
1. ChunkID (PK) 2. FileID (FK) 3. S3Link

Deep Dive

How User Can View The Old Version Of File

We will keep the info for old chunk and the new chunk that is updated, sorted by created data.
If a user want to a file version that was created on x date, we will map chunks that were created on x date

Database

As consistency is very important, its better to use RDBMS, but how we gonna scale that?

Design Review By ChatGPT

Your system design for a Dropbox/Google Drive-like service is well-structured and demonstrates a solid understanding of distributed file storage, synchronization, and scalability.

Below is my detailed review:

Strengths

1. **Clear Functional and Non-Functional Requirements:**
    - Well-defined operations (upload, update, sync, version control, sharing).
    - Focus on low latency and high consistency.
2. **Good API Design:**
    - Basic file operations (`POST /upload`) are covered.
    - Could be improved by adding APIs for versioning, file retrieval, and notifications.
3. **High-Level Design and Flow:**
    - The initial design covers the overall flow well.
    - The revised version significantly improves efficiency by:
        - Using **chunking** to handle large file uploads.
        - Introducing **hashing** for deduplication.
        - Avoiding redundant file uploads.
        - Using **queues** to handle offline clients.
4. **Efficient File Storage Strategy:**
    - **Amazon S3 integration** for scalable file storage.
    - **Chunk-based file storage** to optimize upload/download efficiency.
    - **Hash-based deduplication** for storage and bandwidth savings.
5. **Handling Offline Clients:**
    - Queue-based updates ensure devices receive updates once they are online.
6. **Versioning Support:**
    - Maintaining previous file chunks instead of full copies is an efficient approach.
7. **Database Schema:**
    - Good normalization, ensuring file and chunk relationships are well-defined.
    - **Use of Foreign Keys** to maintain relationships.
    - **Tracking devices per user** for better sync management.

Areas for Improvement

1. API Design Enhancements

- **Support for Partial Uploads**
    - Instead of `POST /upload`, implement chunked uploads:
        - `POST /upload/initiate` → Get S3 pre-signed URL.
        - `PUT /upload/{chunk_id}` → Upload individual chunks.
        - `POST /upload/complete` → Finalize file upload.
- **File Retrieval API**
    - `GET /file/{file_id}` → Get file metadata and download URL.
- **File Versioning API**
    - `GET /file/{file_id}/versions` → Fetch previous versions.
- **File Sharing API**
    - `POST /file/{file_id}/share` → Share with users, set permissions.

2. Improving Scalability of Metadata Storage

- **Scaling RDBMS**
    - Use **Sharding (e.g., User ID-based partitioning)** to distribute load.
    - **Read Replicas** for high read traffic.
    - **Caching (Redis)** for frequently accessed metadata.
- **Metadata in NoSQL**
    - If user base grows significantly, consider **NoSQL (DynamoDB/Cassandra)** for file metadata.

3. Handling Concurrent Edits

- **Optimistic Locking**
    - Use **Version IDs** for file updates to prevent overwrites.
- **Merge Conflict Resolution**
    - Implement **client-side conflict resolution (like Google Docs)** or **server-side merging**.

4. Notifications and Sync Enhancements

- **WebSockets or Server-Sent Events (SSE)**
    - To improve real-time sync instead of polling.
- **Client Sync Optimization**
    - Instead of full re-download, **implement delta sync (Google Drive does this)**.

5. Security Considerations

- **Authentication & Authorization**
    - OAuth2/JWT for API access.
    - Enforce **role-based access control (RBAC)** for shared files.
- **Data Encryption**
    - **Encrypt files at rest (S3 default encryption, KMS)**.
    - **Encrypt file transfers using TLS**.

Final Thoughts

Your design is well thought out and aligns with industry practices. The main areas for improvement include:

1. **Better API granularity for uploads, versioning, and sharing.**
2. **Scalability improvements for metadata storage (sharding, caching).**
3. **Conflict resolution for concurrent edits.**
4. **Real-time sync improvements using WebSockets or Delta Sync.**
5. **Enhanced security (RBAC, encryption, authentication).**