Let’s dive deep into **Dropbox / Google Drive system design**, focusing on core architecture, components, scalability, and interview-specific insights.

**☁️ Deep Dive: Dropbox / Google Drive System Design**

**✅ 1. Functional Requirements**

* Upload/download files (any size)
* Organize files into folders
* Share files (public/private links, permissions)
* Edit/collaborate (Docs, Sheets — optional)
* Sync across devices
* Version history
* Search files
* Offline access

**🔒 Non-Functional Requirements**

* High availability and durability
* Strong consistency (especially for sync)
* Low latency (upload/download)
* Multi-device sync
* Secure access (auth, encryption)
* Scalable to billions of files and users

**🧱 2. High-Level Architecture**

sql

CopyEdit

+------------+ +-----------------+

| Client | <----> | Sync Service |

+------------+ +--------+--------+

|

+-------------------------+---------------------+

| | |

+--------v---------+ +---------v---------+ +-------v--------+

| File Metadata | | File Storage | | Notification |

| Service (DB) | | (Chunks in S3/HDFS)| | Service |

+--------+---------+ +---------+---------+ +----------------+

| |

+-------------v-------------+ +-----v---------+

| User & Auth Service | | Versioning DB |

+--------------------------+ +---------------+

|

+--------v---------+

| Sharing Service |

+------------------+

**🗂️ 3. Core Components**

**📁 A. File Metadata Service**

Stores file/folder hierarchy, ownership, and references to chunks.

**Schema (SQL/NoSQL):**

sql

CopyEdit

Table: files

- file\_id

- user\_id

- parent\_folder\_id

- file\_name

- is\_folder

- size

- last\_modified

- chunk\_ids[] (ordered)

- version\_ids[]

Supports:

* Pagination, file tree traversal
* Sharing metadata (access control)
* Search metadata indexing (name, tags)

**📦 B. Chunked File Storage**

Files split into **chunks** (e.g., 4MB) → each chunk stored in object store (e.g., **S3**, **HDFS**, or custom blob store).

**Benefits:**

* Upload resumption
* Deduplication (same chunk, multiple files)
* Partial sync efficiency

**Chunk Schema:**

sql

CopyEdit

chunk\_id: hash(sha256)

storage\_path: URI

ref\_count: int

**🔄 C. Sync Service**

* Watches file system changes (client agents)
* Debounces rapid changes
* Resolves merge conflicts
* Pushes deltas via WebSocket or long-poll

**Design Options:**

* State-based (periodic polling)
* Event-based (client-generated events)

**Interview Question:**

How do you sync a file edited on two devices at the same time?

**🔁 D. Versioning Service**

Tracks file versions, supports rollback.

**Table: file\_versions**

sql

CopyEdit

version\_id

file\_id

chunk\_ids[]

timestamp

committed\_by

Stored in metadata DB or NoSQL (Cassandra, DynamoDB).

**🧑‍🤝‍🧑 E. Sharing & Access Control**

Controls access to files via:

* Public URLs (tokenized)
* User-to-user sharing
* Org-level permissions

**Security:**

* OAuth2 / JWT
* Signed URLs with expiration for public links

**🔔 F. Notification Service**

* Used for multi-device sync alerts
* Kafka + WebSocket push
* Handles:
  + File updates
  + Shared access events
  + Mention tags (Docs style)

**🔍 G. Search Service**

Index file names, tags, content (if allowed).

* ElasticSearch or Lucene-based
* Tokenized + filtered
* Full-text + metadata search

**🔐 H. Authentication & Authorization**

* OAuth2 / SSO for enterprise
* Fine-grained RBAC (file/folder level)
* Session management + MFA

**📤 4. File Upload Flow**

1. Client splits file into N chunks
2. Chunks are hashed, uploaded in parallel to object store
3. Client sends metadata (chunk IDs, file tree info)
4. Metadata service commits file & version record
5. Notify other devices (WebSocket)

**Resume upload:**

* Only re-upload missing chunks

**📥 5. File Download Flow**

1. Client requests metadata → gets ordered list of chunk IDs
2. Fetch chunks from object store (parallel)
3. Reassemble and serve to user

**Optimization:**

* CDN for popular chunks/files
* Range-based HTTP download (streaming)

**⚖️ 6. Deduplication & Storage Optimization**

* Use hash (SHA256) of chunk
* Store once, reference many (ref counting)
* Prevent abuse via rate limits + quotas

**🧠 7. Trade-offs & Interview Questions**

| **Topic** | **Discussion** |
| --- | --- |
| Chunking | What’s ideal chunk size? Trade-offs between overhead and parallelism |
| Consistency | How to handle race conditions in folder hierarchy |
| Sync Conflicts | How to resolve when same file edited on two devices |
| Deduplication | How to detect duplicates securely without violating privacy |
| Access Control | How to revoke access to shared file? |
| Rate Limits | How to avoid DDoS via file uploads |
| Metadata Sharding | Shard by user\_id or file\_id? |
| Offline Edits | How to merge offline edits after reconnection? |
| Atomic Operations | How to move folders/files atomically across users |

**🚀 8. Scalability Strategies**

| **Component** | **Strategy** |
| --- | --- |
| Chunk Store | Use S3 + CDN (CloudFront) or internal blob store |
| Metadata DB | Partition by user\_id, use GSI for path-based queries |
| Sync Service | WebSocket sharded by user |
| Notification | Kafka topics partitioned by user |
| Versioning | Append-only model, TTL for old versions |
| Search | Shard ElasticSearch by org\_id / user\_id |
| Deduplication | Hash-index DB, LRU cleanup |

**🛡️ 9. Security**

* End-to-end encryption (optional)
* AES-GCM encryption at rest
* TLS in transit
* Audit logs for sensitive file access
* Access scopes + time-bound links

**💡 10. C++ Component (Optional)**

cpp

CopyEdit

class ChunkUploader {

public:

ChunkUploader(S3Client& client);

bool uploadChunk(const std::string& filePath);

std::string computeHash(const std::string& chunkData);

private:

S3Client& client;

};

Would you like diagrams for **sync**, **sharing**, or **upload/download pipelines**? Or a real-time conflict resolution deep dive?

**You said:**

Yes please

**ChatGPT said:**

Getting started

**You said:**

real-time conflict resolution deep dive

**ChatGPT said:**

Real-time **conflict resolution** in systems like **Dropbox** or **Google Drive** is critical when **multiple users or devices edit the same file concurrently**. Here's a deep dive into **strategies, components, and trade-offs** relevant for **system design interviews**:

**🧩 Problem**

* Multiple users or devices **edit or sync the same file** at the same time.
* Need to **detect conflicts**, **resolve them**, and **preserve user intent**.

**⚙️ Core Components Involved**

pgsql

CopyEdit

+------------+ +---------------+ +-------------------+

| Clients | <---> | Sync Service | <---> | Metadata Service |

+------------+ +---------------+ +-------------------+

| | |

| v v

| +------------------+ +---------------------+

| | Chunk Store (S3) | | Conflict Resolution |

| +------------------+ +---------------------+

| |

v v

Notification Svc ------------------------> User Resolution Workflow

**🛠️ Conflict Detection Strategies**

**1. Last Write Wins (LWW)**

* Simplest: server timestamps each update.
* Whichever write comes last wins.
* **❌ May lose user data**, bad for real-time editors.

**2. Version Vector / Generation ID**

* Each file has a **version ID** or **vector clock**.
* On sync/upload, version is checked.
* If version mismatch → **conflict detected**.
* More robust than LWW.

**3. Hash Comparison**

* File hash is compared on sync.
* Mismatch = conflict.
* Used to detect binary-level differences.

**🔄 Conflict Resolution Strategies**

**✅ 1. Automatic Merge (for structured files)**

* For text, code, or JSON:
  + Use **3-way merge**: base, local, remote.
  + Keep changes from both ends if possible.

**🧍‍♂️ 2. User-Assisted Merge**

* For binaries (e.g., Word docs, videos):
  + Detect conflict.
  + Store both copies:
    - File.docx (User1)
    - File.docx (User2’s copy)
  + Prompt user to pick.

**⚖️ 3. Operational Transform (OT) / CRDT**

* Used in real-time collaborative editors.
* OT: Transform operations (insert/delete) to maintain consistency.
* CRDTs: Conflict-free replicated data types.
* Ideal for live Google Docs–like editing.
* **Complex but robust.**

**🔁 File Versioning**

Each file has:

* **Current version pointer**
* **History of previous versions**
* Can store:
  + Full copies (for small files)
  + Or **delta diffs** (patches) to save space

**Schema:**

sql

CopyEdit

Table: file\_versions

- file\_id

- version\_id

- timestamp

- parent\_version

- delta (optional)

- storage\_pointer

**📥 Conflict Flow Example**

1. **Client A** starts editing doc.txt (version 5).
2. **Client B** edits doc.txt simultaneously (also version 5).
3. Both sync to server:
   * Server sees **version conflict**.
   * Creates two versions:
     + A’s becomes version 6.
     + B’s becomes version 6a (conflict).
   * Notifies both clients of conflict.
4. **User merges or picks preferred version**.

**📡 Real-time Sync Architecture**

* Sync daemon tracks file changes.
* On change:
  + Compute diff/hash.
  + Check server metadata.
  + If version mismatch → trigger conflict handler.
* Sync service can use **Kafka topics per file or user**.

**🚨 Interview Questions**

**🧠 High-Level**

* How would you detect and resolve sync conflicts across devices?
* What’s the trade-off between automatic merge vs user-assisted?

**🔍 Specific**

* How would you ensure atomic file updates across replicas?
* Design a system that supports real-time co-editing and syncing.
* How does Dropbox preserve both copies during conflict?

**🧠 Trade-offs**

| **Strategy** | **Pros** | **Cons** |
| --- | --- | --- |
| LWW | Simple, fast | Risk of data loss |
| Versioning | Reliable, audit trail | Storage cost |
| 3-way merge | Balances automation/user intent | Not great for binary files |
| CRDT/OT | Best for real-time collaboration | Complex to implement |