implementation

Type

- files to fixed size chunks
- each chunk is identified by a hash (SHA256)
- Client side Metadata →
  - Dir-level →
    - Dir ID : path from the root dir
    - Dir name
    - list of Dir : Dir IDs
    - List of files : File IDs
  - File-level →
    - File ID : Path from the root dir
    - Name : File Name
    - Type : File type
    - Version Number : Each time file is updated we increment the version
    - Timestamp: last modified timestamp
    - **Overall File Hash:** An aggregated hash (or a manifest hash) computed from individual chunk hashes for integrity verification.
    - List of Chunks : list with chunk id
    - status: uploading/complete
  - Chunk-level imformation →
    - chunk id : hash (SHA256)
    - Offset : Order of the chunk in the file

        - Size / length

- Server side Metadata →
    - Dir-level →
        - Dir ID : path from the root dir
        - Dir name
        - list of Dir : Dir IDs
        - List of files : File IDs
    - File-level →
        - File ID : Path from the root dir
        - Name : File Name
        - Type : File type
        - Version Number : Each time file is updated we increment the version
        - Timestamp: last modified timestamp
        - **Overall File Hash:** An aggregated hash (or a manifest hash) computed from individual chunk hashes for integrity verification.
        - List of Chunks : list with chunk id
        - **status: available/deleted**
    - Chunk-level imformation →
        - chunk id : hash (SHA256)
        - Offset : Order of the chunk in the file
        - Size / length
        - State: Download/available/deleted
        -

- Local Changes detection

- Prepare Update Request
- Upload and synchronization

Conflict Detection and Resolution

Case 1: Two Users Update Simultaneously (Concurrent Uploads)

- **Scenario:** Both users start editing the same file based on the same base revision. User A and User B both generate updates (modified chunks) and send them to the server.
- **Server Detection:**
  - The server checks the base revision included with each update.
  - When the first update (say from User A) arrives, it is accepted, and the server increments the file's version.
  - When the second update (from User B) arrives, the base revision no longer matches the current version on the server.
- **Resolution:**
  - **Automatic Conflict Handling:** The server can choose to create a "conflicted copy." This means it saves User B's changes as a separate file (for example, renaming it to something like `filename (conflicted copy YYYY-MM-DD).ext`), thereby preserving both versions.
  - **Notification/Manual Merge:** Alternatively, the server might reject the second update, prompting User B to manually merge the changes based on the updated file. The decision depends on your design requirements.

Case 2: Update Broadcast and Subsequent Local Edit

- **Scenario:** The server broadcasts an update to all connected clients. Suppose User A's update reaches User B's machine, but before syncing, User B edits the file locally.

- **Conflict Detection:**
  - **Local Version Mismatch:** When User B tries to sync, their client compares the local metadata (which now has un-synced changes) against the latest version broadcast by the server.
  - **Version/Revision Check:** The client sees that its current working version does not match the version on the server (e.g., different revision numbers or mismatched chunk hashes).
- **Resolution:**
  - **Local Conflicted Copy:** Similar to the concurrent update case, the client can automatically create a “conflicted copy” locally, preserving both the new local changes and the downloaded version from the server.
  - **Merge Assistance:** In some cases, if changes are on different chunks or can be merged automatically, a merge might be attempted; however, if an automatic merge is ambiguous, creating a conflicted copy is a safe fallback.

BASE version, Local version, and remote version ... SHOULD WE MAKE A BASE COPY EVERY TIME WE EDIT ?

No, At the time of conflict User B will change the file name to conflicted file.

As server has already both the local and remote version it will compute the patch and send to User B so it can generate the lasted copy from the local copy without downloading the full file.

Patch Generation:

Alternatively, the server (or client) can compute a patch—essentially a set of instructions that transform the local base version into the new version. This way, User B only downloads the patch, which is typically much smaller than the full file.

root_dss → root directory

API LIST →

DIRECTORY

/create_directory/{dir_id} → 200OK

/delete_directory/{dir_id}

/list_directory/{dir_id} → {dir:{list},file:{list}}

FILE

first 2 same as directory

/download_file/{file_id} → {list of chunkid}

/download_chunk/{chunk_id} → { metadata: { } , data : {} }

UPDATE FILE

/update_request/ {file_version} & {file_metadata & chunkIDs} → {nb:{list of chunks}} (nb→ need blocks)

User A

server side version number increased to block any other update [status : downloading]

client side version number increased [status: upload]

n number of chunks at a time

/store_chunk/ [{chunkid, metadata:{} , data: {} }] → 200 ok

on receiving the chunk update chunk status

/commit_update/ { file_metadata & chunkIDs } → 200ok if all chunk receive otherwise send chunkid

User B

server blocks the request, resolves according to Case 1

response : { status: conflict , Patch : { action : { insert/delete} , chunk: { metadata, data }}}

User B renames the Local version as conflicted copy.

User B generated the remoted version using the Patch .

DELETION

/delete_directory/{dir_id} → 200 ok

/delete_file/{dir_id} → 200 ok