**General Client Flow**

1. Client refresh timeout expires.
2. Client FileNodeMonitor scans directory tree for directory/file changes. Client asks server for list of updated files.
3. Modified files are added to a watch list.
4. When a file in the modified-file watch list hasn’t been modified for a while (we can work with the file without user interruption), ask the server for changes to that file (update before commit). If the files has changed, backup the file and then Rsync the server changes into the file without destroying the new content (is this possible? Actually, it seems unnecessary since the modified blocks are already on the server; that’s the whole point of rsync afterall. But how do we retain and merge client data?).
5. When server file changes have been merged, segment the file and send it to the server. Segment one file at a time. When the segmenter has finished segmenting the file, the segment list is sent to the assembler and the segment report (bit packed version of the segment list) is sent to the server.
6. Server hashes the segments and segments the server file version. Server compiles instruction list containing server data segments and client segment IDs (when a match is found). Packed reassembly instructions are sent back to the client.
7. Client reassembles file, per the server instructions, into a stage.
8. When the stage is completed, the Assembler checks whether the file on disk has been changed locally since it was segmented. If it has, repeat #4 in the opposite direction (wait until the user is no longer modifying the file; note that this also naturally means that we’ll have to send the updates to the server). Otherwise, lock the file and push the stage to disk as quickly as possible.

Other thoughts:

* To find out about changes, client has to ask server. Server replies with list of changed file ids. If the client doesn’t have the changed file, it asks the server for the entire file. If the client has an old, unmodified version of the file, it initiates a classic rsync transaction.
* If the client version of the file has been updated, but the server version has not been modified, the client initiates a reverse version of the rsync transaction.
* What do we do if both files have changed? Do we need to enforce in order updates (i.e. the client that modified logically first will merge with an earlier version of the file, which is then merged with more logically-recent updates from another client?)

CRUD scenarios:

* New file at client:
  1. When changes to the file have settled out, client sends Create message containing file name and path. Server knows, based on path, who else is allowed to see the file.
  2. Server checks for existing entries with the same file/path within the user and group.
     1. File already exists:
        1. Create new entry, but append “\_conflict” before the file extension.
        2. Server sends Conflict notification to client.
        3. Client adds new name to ignore list (so that the FS monitor does not report it as a new file again) and the previous name to the update list (since it apparently didn’t have the original before).
        4. Client renames the file to the name generated in step (1).
     2. File does not already exist:
        1. Otherwise, server creates entry for file in DB indicating file name/path, owner, group ID, current update clock (logical clock), and timestamp (but whose timestamp?).
        2. Add Create entry (first entry for any file) for this file in the Revisions collection.
  3. Server sends an empty segment report (just the header) to the client as if it was just trying to get changes for the file.
  4. Client sends an Assembly Instruction list containing only Chunk instructions.
* File is updated at client:

1. When changes to the file have settled out, client sends Update notification to the server.
2. Server checks for update conflict using vector timestamps. Since users are not allowed to simultaneously edit files, the server must compare vector timestamps to determine if a conflict exists. The first client to update a file wins
   1. Conflict exists:
      1. If the server detects a conflict after receiving an update notification, it performs a procedure similar to (New File: b.i) above.
      2. Create new entry in group collection, but append “\_conflict” before the file extension.
      3. Server sends Conflict notification to client. Client renames local file.
      4. Server sends empty segment list for conflict file.
      5. Client sends Assembly Instruction list containing only chunks.
      6. Server assembles conflict file.
      7. Server adds entry to Revision collection to show that the file was branched due to a conflict.
   2. No conflict exists:
      1. Server sends segmented file to client.
      2. Client acts as file authority and sends Assembly Instruction list to server.
      3. Server assembles file and adds an entry to the Revision history collection for the file.

* Client has not changed any files, but changes exist at server:

1. Client sends periodic requests for changes to files in its group.
2. Server checks last update time for user and then reads the Revisions collection to find the first change since the retrieved last update time. More specifically, if there are multiple updates to the same file since the last update time, the server picks the newest.
3. Server sends a change notification for the single file to the client.
4. Client checks the type of change notification:
5. If the change notification is a Delete notification, the client removes the file.
6. If the change notification is an Update notification, Client segments the local file and sends it to the server.
7. Server acts as file authority and sends assembly instruction list to client.
8. Client reassembles the file and immediately sends a new update request to the server.
9. Server increments update clock and repeats from (b).
10. (a) through (g) are repeated until the clients update clock is current and then returns to sending periodic requests.

* Client deletes a file:

1. The file system monitor detects the deletion of a file and sends a Delete request to the server.
2. Server adds Delete record to the Revisions record for the file, but the file is not actually deleted Subsequent update requests from other clients will encounter the Delete record as in (d.1) above.

Users Collection:

|  |  |
| --- | --- |
| Field Name | Description |
| userId | Auto-incrementing unique user ID. |
| lastName | User’s last name |
| firstName | User’s first name |
| groups | List of groups that the user is member of. Groups are synonymous with root directories. |

Groups Collection:

|  |  |
| --- | --- |
| ID |  |
| Owner ID |  |
| Admin list |  |
| Root hash |  |
| Current Revision # |  |

Group Changes Collection:

|  |  |
| --- | --- |
| Change revision # |  |
|  |  |

Revisions:

|  |  |
| --- | --- |
| Type | Create, Update, Delete |
| File ID hash |  |
| File revision number |  |
| Group revision number |  |