Distributed Storage Service

The distributed storage service we will build will have two types of users - 1. Service User, 2. Admin
The codebase will be specifically for running on **Linux** Devices.

Features:

The system that we plan to build will have a similar user interface similar to Dropbox.

1. The service will work on a single directory on one of the user's devices. For simplicity, we call this directory `root_dss` in the rest of the document. This service will recursively track all **regular files** and subdirectories under this directory.
2. The user can use this directory similar to any other directory in the Linux device. More specifically, one can do the following.
    a. Create a regular file.
    b. Delete a regular file.
    c. Modify any regular file. (How the user modifies the file is irrelevant here. Whether using a terminal-based editor or some other tools, what matters is the content of the file being modified)
    d. Rename a regular file. (This happens only when the files before and after rename are in the subtree of `root_ds` in the original filesystem tree. Otherwise, it is case (a) or (b).)
    e. Create a new directory.
    f. Delete an existing directory along with all the files in it
    g. Rename a directory
    h. Update metadata of a regular file or a directory
3. The system will keep the contents of `root_dss` synchronized with the backend Distributed Storage Server.
4. To use the service, the user first needs to create an account, log in, and register his account. After this, the client executable needs to be run by the user. (We also plan to run the client as a daemon that automatically starts on boot using Linux's service managers without manual intervention). The user can log in from different devices and register different `root_dss` on each of them. However the system will synchronize the contents of all such `root_dss` of a single user. More specifically, the client will operate as follows:
    a. On boot, check for changes in the local `root_dss` and at the server. If there are no conflicting changes, then receive/send the changes. Otherwise, create a conflicting copy and expect the user to notice the conflicting copies and keep the one as desired.
    b. After boot, it will continuously monitor `root_dss` for any changes made by the user, send the corresponding changes to the server and also listen for any incoming changes from the server. Depending upon the presence of the conflicts, either a conflicted copy will be created, or an automatic merge will be performed.

Admin Functionalities:

1. An admin will be able to do the following:
    a. Monitor system conditions: View node health, storage usage, logs
    b. Get the list of users
    c. Create, delete, update users, set permissions(read/write)
    d. Add/remove servers when faults happen when a server needs to be updated(software/hardware updates), or for other reasons.
2. For the **Consistency model on the client side,** we are thinking of using a hybrid of :
    a. **Monotonic Read Consistency**: Ensures that once a user reads a version of a file, they won't see an older version later.
    b. **Eventual Consistency**: Guarantees that all replicas will **eventually** converge to the latest state

Server Side Design:

For server-side distributed storage, we intend to run Ceph (or a similar service) on three devices and simulate distributed storage

Bratin Mondal
21CS10016

Swarnabh Mandal
21CS10068

Soukhin Nayek
21CS10062