Elephant (name TBD) is a system that lets people donate hard disk space to a server operator (e.g. archive.org, libgen, sci-hub, etc). It works over torrents, in a server-client architecture. The server prepares the torrents, and the clients tell the server how much space they have. The server then gives the clients torrents to download and seed. The client passes these torrents to a torrent program (e.g. Transmission) through the torrent program's API. The client is responsible for managing the torrent program.

To run the client and donate space, you specify the server you want to subscribe to, and how much space you want to give it. You also possibly specify a path under which to store the files locally.

(e.g. elephant https://elephant.archive.org 100G /home/stavros/elephant_data)

Elephant runs as a daemon, so the above will probably live in a config file somewhere, so you can add multiple entries and servers.

Every so often, the clients make a request to the server's API, sending the following information:

- The client's unique ID
- A list of torrent IDs that the client has
- A list of percentages of completion of those torrents

The server replies to the client with a list of torrents to download and seed. If the client already has some torrents, the server tries to keep the new list fairly similar to what's already on the client.

When the client receives the list, it performs the following steps:

- Adds any torrents from the response that it doesn't currently have
- Removes any torrents that it currently has but that are not in the response
- Checks the files in the local file system and the files in all the torrents, and deletes the files from the former that are not in the latter.

The files that the torrents contain are organized in the following manner: To create a torrent, the server operator gives the server a list of files. The server calculates the hashes of those files, and creates a hierarchy of, if the hash is e.g "123456789abcdef", "12/1234/123456789abcdef". So, each file goes in a folder of its first two hash characters, then a subfolder of the first four hash characters, and the filename is the full hash. This allows the clients to only download each file once, and share it between any number of torrents. It also allows the server operator to replace a torrent with a superset, containing e.g. more episodes of a series, without the clients having to redownload the existing episodes.

Along with this, a text file is included in the torrent, called `/<torrent_id>.toml`, containing the mapping between the path and the original filename, like so:

```
12/1234/12345678abcdef=Charles Dickens - Great Expectations.pdf
```

The server also has a notion of a “successor” torrent, so, for example, if a series of books has nine items, and a tenth book comes out in the series, a new torrent can be created with all ten books, as a successor of the previous torrent with nine books. The server will know to prioritize replacing the old, superseded torrent with the new successor whenever possible (whenever space permits).

The server assigns files to clients by solving the knapsack problem, where the value of each torrent is its rarity, and the size is, well, its size.

To avoid some attacks, the value should probably be a probability that each item has to be assigned. This is to avoid an attack where a botnet claims to have a torrent (that the botnet wants deleted), to trick the server into not assigning it to anyone. If the probability of each item is nonzero, it can still be assigned, even if it’s very popular. This probability will need to be tuned, so popular items still have a chance to be assigned, but not so big that they overtake actually rare torrents.

We probably shouldn’t call these “torrents”, but something like “datasets”, because that’s what they are.

The client only talks to the operating server, it won’t be like IPFS where you can request stuff from your local node and you can read it from local storage directly. Only the server operator is meant to talk to the clients and request things, really.

The server database should have a list of torrents, their sizes, maybe the files they contain. It should also have a list of clients, how much space they are donating, and what torrents they have been assigned. The clients still send all the details above, just to make the protocol a bit more stateless, but the server is free to ignore the info.

The idea is that the entire service can be replicated by just sharing the table of torrents, and each client can be made to talk to the replica, report what they have, and the table of clients will be recreated from those check-ins.

When the client polls, the server can tell it to perform one of a few actions:

- Add a dataset to its local storage.
- Remove a dataset from its local storage.

Replacing an existing dataset is merely removing one dataset and adding another, since the client will reconcile everything after it’s added/removed things, and will keep what files it needs to keep and delete what files it needs to delete.

Maybe the server should be able to tell the clients to switch to another URL, but we need to be careful of attacks where a malicious user tells everyone to switch to a URL that deletes all their data. Then again, if they can tell them where to switch, they can just tell them to delete everything.