DESIGN OF THE IMPLEMENTATION

Names: 1. Sai Raghu Vamsi Anumula (UFID: 49939544)

2. Srikar Choppakatla (UFID: 54983368)

MONGO DB:

We have used PyMongo for this as we are using python for working with MongoDB.

The first step is to create a MongoClient instance to the running mongo DB server.

The file system is implemented as with Monogclient.database.collection data structute. In which each key (filepath&&<field_name>) has a unique document.

The APIs being exposed by the PyMongo like find_one(),update() are being used in the get and put methods to write, read values from the database.

Pseudo code:

To ensure that the state of file system is restored as per the database, the init method has to be modified to skip initialization of fileNode fileds on every client mount.

Class FileNode

```
{
If not fsys.find_one({"key":<filename&&meta>}) If the metadata is not found in the DB
        self.put("meta",{})
If not fsys.find_one({"key":<filename&&listnodes>})//
        self.put("list_nodes",{})
If not fsys.find_one({"key":<filename&&data>})//
  self.put("data","")

}
```

The above code ensures that the file contents are not lost when we restart the database.

Put()

fsys.update(key,value) // Update FS db with the key and marshalled value

GET()

File_dict = find_one(key) // Fetches the File dict in database with the key specified

If (key) – If the key is found

return pickle.loads(file_dict[“value”]) //unmarshalling the data and reading it

else

return NONE

}

CACHE INTEGRATION WITH THE ABOVE IMPLEMENTATION:

The cache is implemented as an ordered dictionary. The reason for this is because we are using an LRU policy for cache contents replacement.

The cache size parameter is only accessible to the programmer which has a default value of 10.

- When we are putting/writing data, we do it to both the cache and MongoDB.
- When we are getting/reading data, we first look for it in the cache and then if its not found we look for it in the database.

Pseudo code (for the cache class methods):

```
CLASS LRUCACHE
{
init()
        Self.cache = collections.OrderedDict()//cache is implemented as an ordered dict

Get()
        value = self.cache.pop(key) // gets the value corresponding to key in the cache
        If  [value is not empty]
        return value
        else
         mongodb.get() //Fetch data from DB

Put()
        try:
        self.cache.pop(key) // Removes if there is value with same key
      except KeyError:
        if len(self.cache) < self.size: //if cache is not full it inserts the new element
           self.cache[key] = value
        elif len(self.cache) == self.size: // if the cache is full
           self.cache.popitem(last=False)// pop the Least recently used(LRU) item
           self.cache[key] = value
}
```

TEST SCRIPTS:

MONGO WITH CACHE:

To mount the filesystem using the python file, the syntax is (assuming it has a default port no. 27017)

Get and put are modified to print the computation times of those methods.

Host$ python mongo_cache.py mountdir IPAddress

Ex;- Host$ python mongo_cache.py fusemount 127.0.0.1

MONGO WITHOUT CACHE:

To mount the filesystem using the python file, the syntax is (assuming it has a default port no. 27017)

Host$ python MongoRFS.py mountdir IPAddress

Ex;- Host$ python MongoRFS.py fusemount 127.0.0.1

We have tested the implementation using various FS operations.

We have also written 2 shell scripts to test the performance. We have two scripts test-dir.sh and test-files.sh.

Test-dir.sh –

Usage:

$ sh test-dir.sh <path_to_mount_root_dir>

This script creates a directory in the mounted Directory and changes directory to the new dir. And It prints runtime for changing the directory.

In the presence of cache it shows improvement in run time performance.

Test-files.sh

Usage:

$ sh test-files.sh <path_to_mount_root_dir>

This script creates files and prints the contents. The performance improvement using cache is seen in terms of running time in printing the file contents.

We have tested these using a MongoDB server running on Amazon WS Ubuntu instance.

We observed 10 times improvement in the runtime performance.