EEL-4736/EEL-5737 PoCSD - Final Project

Assigned: 10/25/2019; Final code and report due on 12/4/2019

To be done individually, or in groups of two

In the last assignment, you built a client/server based file system where the client accessed data blocks stored at a server. In this final project, you will extend your file system to multiple servers with support for redundant block storage. The project will expose you to practical issues in the design of systems including: client/service, networking, and fault tolerance. Your design must work with at least 4 and up to 16 servers (the file servers may run on the same computer). You can use your client/server file system of homework # 4 as a starting point for this project.

Your goal in the project is to distribute and store data across multiple data servers to 1) reduce their load (i.e., distributing requests across servers holding replicas), provide increased aggregate capacity, and fault tolerance. Your redundant block storage should follow the general approach described for RAID-5. Use integers to identify your servers, and configure the system so that there is a total of N=4 servers. You need to distribute data and parity information across servers, at the granularity of the block size from your configuration file.

For reads: when there are no failures, your design should allow for load-balancing, distributing requests across the servers holding data for different blocks – i.e., for a large file consisting of B blocks, you should expect to have on average N/5 requests handled by each server. For writes: writes should attempt to update both the data and parity block. If acknowledgments from **both** replicas return, your write can complete successfully and the client returns. If only **one** acknowledgment is received, your system should register the server that did not respond as failed, and continue future operations (reads/writes) using only the remaining, non-faulty servers. Your design should be able to tolerate a fail-stop failure of a single server, but you do not need to implement the process of repair and recovery when a server crashes.

Required for EEL5737 students; not required for EEL4736 students: conduct a performance analysis of your design, comparing quantitatively the average load (i.e. number of requests handled per server), with and without failure, of your design compared to the baseline case of single-server (homework #4).

Extra credit for all EEL5737 and EEL4736 students: for extra credit, you may 1) implement an additional design that follows the RAID-1 approach, and configure via command line whether your system works in RAID-1 or RAID-5 mode, and/or 2) implement the ability to tolerate decay in a stored block in addition to fail-stop. For 2), your system needs to allow you to corrupt a block in any of the servers, and show that corrupt blocks can be detected and corrected.

Note on extra credits: extra credit 1) will add up to 50 points towards your grade in the programming part of HW#2, and extra credit 2) will add up to 50 points towards your grade in programming part of HW#3

Expectations on how your system should operate:

Client interactive window:
Your client should take simple commands interactively from the terminal. Your client should parse the commands, and map them appropriately to which function you have to call, passing its respective parameters. The output should be printed on the screen

- Use the dollar sign "$ " (with space after it) as a prompt
- Once started, the client takes one command at a time (one per line), until entering the "exit" command, which terminates it.
- You must implement the following commands: mkdir, create, mv, read, write, status, rm (all lowercase)

For example:
$ mkdir /A/B
$ create /a.txt

Write:
Before a write, your system should display the server identities on which you will write the data. After a programmable delay (entered as an argument), you should proceed with the write. During write, after writing to a data server, you should mention which server is being written a parity block, you should provide a wait statement inside your code with the assigned delay which should display in the terminal as "waiting to write parity" with its identity. You should assume that at most one server will be down during writes.

Read:
Before read, your system should display the server identity on which your data resides. After the delay of "5" seconds, you should proceed with the read. Your system should be fault tolerant in a way that, if one of the server fails, it should fetch and reconstruct the data from the other servers. You can maintain a table which will keep track of this. Once your server is down, you can assume that it will not be brought up again.

You are free to modify any part of code with the condition that it should support the same API from the top.