EEL-4736/EEL-5737 Principles of Computer System Design

Final Project

Assigned: 10/25/2017; Due 5 pm on **12/06/2017**

To be done individually or in a group of two

In the last assignment, you built a client/server based file system with one metaserver and one dataserver. You used remote servers - metaserver and dataservers – to store metadata and data of the files/directories respectively. In the final Project, you will extend your files system to store the data in the multiple data servers. The aim is to distribute the data across the multiple servers to reduce their load and address the concepts of distributed systems like fault tolerance and redundancy. You can use your client/server based file system of homework # 4 as a starting point for this project.

In your design, you will need a single metaserver to store the metadata of the files/directories and N dataservers to store file data. Metaserver will be similar to the metaserver you designed in the last homework. While in a total of N dataservers you will distribute the storage load (as evenly as possible) across all the servers. For any file, start storing its block zero at a particular server (to be determined by using a Hash Function). The subsequent blocks need to be stored in a round robin fashion across all the servers, using modulo arithmetic. For example, if hash(path) = x, then blocks can be stored in this manner [x%N, (x+1)%N, (x+2)%n, …] where, each item in the list represents the server number at which a particular data block needs to be stored. The selection of the hash function is left for you. You can use any hash function which serves the purpose of load balancing.

Storing blocks of files in a round robin fashion may lead to collisions. For example, for N=4 block numbers 0 and 4 will be stored in the same function. Your design should take care of resolving such collisions and distinguish the keys in the same server.

Design assumptions and requirements are:

- There are multiple dataservers and a single metaserver. Metaserver is assumed to be very reliable and robust which never fails.
- The data blocks of the file are stored in the round-robin fashion as mentioned above but, there are two redundant copies of the same block are stored in the following two dataservers.
    - Replica 1: [x%N, (x+1)%N, (x+2)%N, (x+3)%N …] *where x is hash(path)*
    - Replica 2: [(x+1)%N, (x+2)%N, (x+3)%N, (x+4)%N, …]
    - Replica 3: [(x+2)%N, (x+3)%N, (x+4)%N, (x+5)%N, …]
    - You are storing 3 replicas of each block means that replication factor is 3.
- The data-servers store the data blocks in persistent storage (hard disk) to allow for recovery upon a crash. The file used to store the data will be referred to as ‘data store’.

    - You can still use an additional in-memory data structure, but you should always write to disk before returning an RPC call. This means that the primary database for the server is on disk. It is suggested that you use Python shelve for a dictionary object backed by a persistent file.

- Your dataservers should tolerate server crashes – where server process is terminated and restarted at later point in time.
    - In case server process crashes and restarts (e.g., server rebooted), the server should be able to recover and resume serving data using the data stored on its local disk.
    - In case persistent storage of the server is completely lost (e.g., server disk is failed and replaced with new disk), then it should be able to recover using the replica(s) rom its adjacent servers and copy the data blocks that are to be stored on the current server.
- Dealing with data corruption – Corruption in data store.
    - Use checksum of data blocks in the data-servers to verify the data.
    - When there is read call from the user, the FUSE client should read from one of the replicas (randomly chosen) and verify the checksum. If the checksum does not match, the client should read from another replica and write back to the corrupt server if the client detects data corruption.
    - Provide an XMLRPC call with name 'corrupt' which takes the path of a file as an argument. The function should simulate the corruption of at least one of the bytes of any of the data blocks (or its checksum) of that file stored on that server.
- (Required for EEL 5737, and optional for EEL4736 students)
  Dealing with writes while a server is unavailable
    - All writes must commit to all replicas and receive acknowledgements from the servers before the FUSE client proceeds further.
    - When server is down, any write calls on the FUSE folder should back. The FUSE client should keep retrying the operation until it succeds and the write call should not return till then.
    - Reads should return successfully even if a single replica is available and the checksum verifies correctly.
- You should test your system with number of data-servers N>=4 and three replicas.

Program arguments and guidelines:

The program should take the arguments in the following format:

python metaserver.py <port for metaserver>

python dataserver.py <0 indexed server number> <ports for all dataservers separated by spaces>

python distributedFS.py <fusemount directory> <metaserver port> <dataservers ports separated by spaces>

example (N=4):

python metaserver.py 2222
python dataserver.py 0 **3333** *4444 5555 6666*
python dataserver.py 1 *3333* **4444** *5555 6666*
python dataserver.py 2 *3333 4444* **5555** *6666*
python dataserver.py 3 *3333 4444 5555* **6666**
python distributedFS.py fusemount 2222 3333 4444 5555 6666

The local ports are bold and adjacent server's ports are italicized for understanding.

Testing:

- Your program will be tested for all the filesystem operations like previous assignments.
- System will be tested for N<=5.
- The cases below will be tested individually. Implies there's no Data Corruption in a server while another server is down.
- Server Crashes
    - Any number of servers crashed (process killed) and restarted with same arguments gain without touching the data tore file.
    - Any number of non-adjucent servers crashed and restarted with the data store file deleted before restarting.
    - On restart, server should come back the same state before crash. Will be tested by retrieving the data after restart.
- Data Corruption
    - One data corruption per server in the same path in non-adjucent servers. E.g., (N=5) server0.corrupt('/file1'); server4.corrupt('/file1')
    - Corruption of different paths in different servers.
    - After corruption, the files will be read from FUSE and verified for correction.
- Server Downtime
    - Same cases as with server crashes, but with read/write during the downtime.
    - The reads should succeed when non-adjacent servers are crashed. Writes should block even if a single server is down.

Submission Guidelines:

Turn in through Canvas following five files (attach individually):

1. distributedFS.py – Client that implements the remote file system with FUSE and XMLRPC
2. metaserver.py – Modified simpleht.py that hosts your metaserver

3. dataserver.py – Modified simpleht.py that hosts your dataserver
4. projectDesign.pdf – PDF document describing the design of your implementation and tests you have conducted to verify the functionality.

Make sure your Python code is well commented and tested before submission. Assignment will be graded on the basis of design and functionality. Copy and the paste the python code (distributedFS.py and metaserver.py, dataserver.py only if changed simpleht.py) at the end of your projectDesign.pdf file. Write your course number (EEL 5737/EEL4736) on the projectDesign.pdf file.