**ECE 428 MP3 --> Team 38: Agnivah Poddar (apoddar3) ; Simran Patil (sppatil2)**

Algorithm used: The operations that this SDFS supports are as follows and are entered as user inputs from any of the VMs. They are then sent to the Master or the currently elected leader to be processed such that total ordering of the operations is maintained. All these operations are handled using TCP except for delete. All SDFS files are saved as "sdfs\_\*" files and the system is cleared of older copies on initialization.

1. put localfilename sdfsfilename: Here, the user puts a local file into the SDFS with the name 'sdfsfilename'. The master hashes the sdfsfilename to 3 non-faulty VMs based on membership list using md5 hashing and a truncating mechanism. We choose to write to 3 machines to maintain a Quorum and handle the case of two simultaneous failures. For handling a Quorum here, we need 5 machines with writes to 3. MD5 ensures that the hash is consistent. An unordered map is used to save the locations at which the key = sdfsfilename is stored. After getting the locations mapped, the master sends the file it received from the client to all the locations. The destination VMs maintain a set of locally present files. We handle conditions such as the client fails while sending the file to master and the operation is aborted and tests to make sure that the requester is running. The user is prompted to provide a confirmation if she wants to write to the same SDFS file within a minute and that confirmation timeouts after 30s.

2. get sdfsfilename localfilename: Here, the user gets a file 'sdfsfilename' from the SDFS and saves it as a 'localfilename' locally. It sends a 'GET sdfsfilename localfilename' request to the master who then checks its File->VM map for the one of the three locations in which it is stored. As soon as it gets one alive location which has the latest update of the file, it sends the IP of that machine back to the requester. The requester machine then makes a connection with this received IP and requests the file, following which the file transfer is initiated.

3. delete sdfsfilename: The user sends a 'delete sdfsfilename' request to the master who handles it by checking its 'File->VM' map. If the file doesn't exist then that message is sent back to user. Else the master sends a 'DELETE' command to each of the locations that are storing a replica of that file, who then delete the file from their respective machines.

4. ls sdfsfilename: The user sends a 'ls sdfsfilename' request to the master who checks its 'File->VM' map to see the locations in which the replica of the file is stored. If that machine is alive, then the master sends back a message to the user, listing the alive machines that have the file.

5. store: Each user maintains a hash set called 'sdfs\_files' that indicates all the filenames in the sdfs that it has stored. When the user inputs 'store', this is handled by printing out the keys stored in this hash set.

We maintain two maps - VM\_to\_Files\_map and master\_file\_directory to map the VMs to the files present and each of the sdfsfiles to the locations where it is present. These are multi-casted to all the VMs every time there is a new file put into the sdfs or a file is deleted or a machine fails. Because all the machines are aware of this metadata about all files and all VM stores, it helps in electing the master on its failure as the member at the beginning of the membership list. When a machine fails or voluntarily leaves, the master hashes the files it had to new locations and fetches the required file from one of the replicas to be sent to the newly chosen destination. Randomness in choosing is maintained using the md5 hashing algorithm with modification and file transfer uses TCP. MP1 helped us grep the data from vm\_log file locally maintained to keep track of 'gossip messages', 'voluntarily leaves' and other such tags that were logged. It helped monitor the VM activities overall.

1) **Re-replication time and Bandwidth upon a failure:**

File Size: 42 MB

This involves fetching the file to the master followed by sending it ahead to the destination. Thus, the total data transfer in the following times involved 84 MB of data. Further latency could be because of the randomness checks used with MD5 and lock acquisition by the thread. These deviations and the causes of latencies are the reasons why we think there were outliers.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Reading 1** | **Reading 2** | **Reading 3** | **Reading 4** | **Reading 5** | **Average** | **Bandwidth** |
| 2 seconds | 3 seconds | 1 second | 7 second | 1 second | 2.8 seconds | 30MBps |

2)**Time to insert, Read and Update**:

The Insert and updates are similar since the design for both are the same. (Both go through a PUT request). Read also performed through master and slightly more due to checking of presence of file.

Case A: 25 MB (All readings are in milliseconds)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Reading1** | **Reading2** | **Reading3** | **Reading4** | **Reading5** | **Average** | **STDev** |
| **Read** | 1991 | 670 | 532 | 1210 | 1009 | 1082.4 | 574.5078764 |
| **Insert** | 1819 | 581 | 415 | 1041 | 991 | 969.4 | 544.4325486 |
| **Update** | 1888 | 485 | 552 | 1001 | 844 | 954 | 563.0785913 |

Case A: 500 MB (All readings are in milliseconds)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Reading1** | **Reading2** | **Reading3** | **Reading4** | **Reading5** | **Average** | **STDev** |
| **Read** | 18119 | 6230 | 4988 | 11078 | 9281 | 9939.2 | 5169.90732 |
| **Insert** | 16007 | 5112 | 3652 | 9160 | 8720.8 | 8530.36 | 4791.044797 |
| **Update** | 16840.96 | 4326.2 | 4923.84 | 8928.92 | 7528.48 | 8509.68 | 5022.661035 |

3) **Time to detect write-write conflicts for 2 consecutive writes within 1 min to same SDFS file:**

These timings are small. This is in accordance to what we had expected as in our design, we maintain a global variable that indicates which sdfs file is currently being out. The conflict is detected after a put as soon as a match is detected and the user is then notified.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Reading 1** | **Reading 2** | **Reading 3** | **Reading 4** | **Reading 5** | **Average** |
| 43403 ns | 50176 ns | 35437 ns | 47031 ns | 37912 ns | 42791.8 ns |

4) Time to store entire the English Wikipedia corpus:

Case A: 4 machines

This case involves the time taken to put the file into the master and then forwarding it to the selected recipients. We hash the file to three destinations and send the file sequentially. This explains why the readings are significantly larger values as we had expected.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Reading 1** | **Reading 2** | **Reading 3** | **Reading 4** | **Reading 5** | **Average** |
| 112 s | 134 s | 119 s | 118 s | 143 s | 125.2 |

Case B: 8 machines

It is similar to the readings for 4 machines since our design for storing files is independent of the number of machines running.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Reading 1** | **Reading 2** | **Reading 3** | **Reading 4** | **Reading 5** | **Average** |
| 135s | 121s | 127s | 149s | 116s | 129.6s |