# Simple Distributed File System
# CS 425 - MP3 - Group 70 (@sushmam3 and @jadonts2)

**Design:** Our distributed file system was built on top of our framework for the MP2 full membership list (i.e. the membership software is the bottom layer of the SDFS stack) and based on a simple implementation of HDFS. At each node, we keep the 4 previous threads (TCP listener, UDP listener, Ping/ACK thread, and command thread), with the addition of one extra thread for the elected coordinator. All messages flow through a single port, and are sent to the correct handler based on type and destination. The introducer is now at a fixed location backed by a file, as per the recommendations. SDFS commands are dispatched to the file server by the command handler thread, and SDFS messages are communicated via TCP.

***FileServer:*** In addition to the setup performed by MP2 when joining, each node running SDFS starts up a file server to handle SDFS messages as well. The file server stores the versioned files, as well as a set of metadata related to SDFS-specific info about each file stored at the machine.

***Coordinator & Election:*** In addition to the original MP2 functionality, we have added support for coordinator election. The coordinator stores metadata about versioned file locations and handles file replication. Most requests to SDFS flow first to the coordinator, then to the relevant nodes. If the process is the first to join the group, it is automatically the coordinator. Otherwise, whenever the failure of the coordinator is detected we use the ring-election protocol to select the most recently joined node (i.e. highest ID), modified slightly to be more resistant to failure. Upon election, the coordinator requests each file server's metadata to rebuild its own metadata.

***Replication:*** Our SDFS system replicates each file version 4 times, so there is always at least 1 copy remaining after 3 simultaneous failures. The coordinator periodically checks for any failures (discovered via MP2), and selects new nodes to store files that need more replicas, then informs existing nodes of the replication request.

***Metadata:*** At each file server, we keep a list of file names delimited by version number stored at that machine. At the coordinator, we store a map, with keys being the SDFS filename, and values being another map with integer version number keys and a set of nodes that store that file version as values: <SDFSFileName, <Integer, Nodes>>.

***Put:*** The file server making the put request first informs the coordinator of the request. The coordinator reserves a new version number and responds with the locations of 4 selected nodes that should store the file. The file server then sends the file to each of these nodes, and informs the coordinator of successes. Even if there are failures, at least one node will receive the file, and the coordinator will then catch the missing copies during replication checks.

***Get & Get-Versions:*** Get works similarly. The file server making the get request first informs the coordinator of the request. The coordinator responds with the locations of the nodes that store the file. The file server then requests the file from the nodes in a random order (to reduce load at any one process), stopping once it has received the file. Note that even if there are failures during this process, at least one node will respond with the file. Get-versions works the same way, but retrieving the *n* (specified via argument) most recent versions instead of the latest.

***ls:*** The ls command sends the same coordinator request as the get-versions command, but instead of retrieving each file, ls prints the list of machines along with the versions they store.

***Store:*** Lists the files stored at this machine as represented in the metadata.

***On Rejoin:*** The directory is cleared each time a process rejoins SDFS after a failure or leave.
***Quorum & Total-Ordering:*** Since all reads and writes are guaranteed to intersect in the coordinator, our quorum value is 1. Only the coordinator needs to ACK reads and writes. Furthermore, even though the TCP message handler handles each message in its own thread, the coordinator's metadata store is a synchronized data structure, meaning that each put will be handled one at a time, getting the next version number. Since the coordinator is the sole decider of file version, total ordering is satisfied.

**Performance Metrics:**

We used MP1 distributed logging to debug issues in the system and to measure the performance metrics.

**i)** To measure the re-replication time and bandwidth on failure for a 40MB file (the only file stored in the system) we used tool nethogs and MP1 logs to view the measured numbers.
For each failure count, we took 5 data points.

| Re- replication 40MB file | Time (in ms) | Bandwidth (MB/s) |
|---|---|---|
| | | |
| 2 failures | | |
| Average | 550 | 150.56 |
| | | |
| 3 failures | | |
| Average | 916 | 137.6 |
| Standard Deviation | 39.87 | 6.058 |

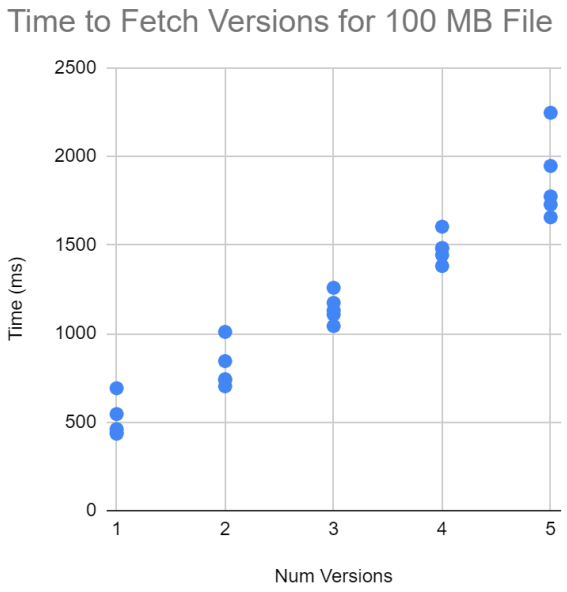
File transfer bandwidth is around 140 MegaBytes/Second. With an increase in the number of failures, more time is required for re-replication.

**ii)** The average time to insert a 25MB file is 751.67 ms whereas a 500MB file is 8831.5 ms. We measured this by timing our program when it receives a command. We have taken 6 data points for each file size. This is expected since we are transferring the file as a complete unit. The time can be improved upon by sharding the file.

| File_Size | Insert (in ms) | Update (in ms) | Read (in ms) | Delete (in ms) |
|---|---|---|---|---|
| | | | | |
| 25mb | | | | |
| Average | 751.67 | 723 | 207 | 14.17 |
| Std dev | 71.17 | 50.1 | 35.3 | 1.33 |
| | | | | |
| 500MB | | | | |
| Average | 8831.5 | 8860.33 | 2932.5 | 49.67 |
| Std dev | 490.02 | 1079.76 | 1607.76 | 18.78 |

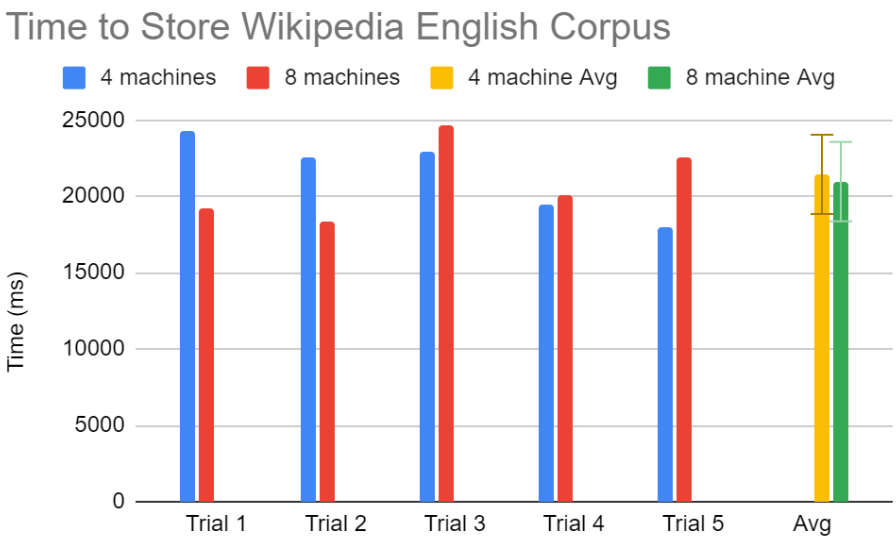
Since updates are treated the same as put except for incrementing the file version number, it takes almost the same time as insert. Average time to update a 25MB file is 723 ms whereas a 500MB file is 8860 ms. Larger file takes more time to read, insert and delete.

**iii)** Since we effectively need a linear increase in fetches when there are more versions, we expect that we will need more time to fetch when there are more versions. We timed our get-versions function using a file of size 100 MB, five times each per number of versions, and observed the following results, in line with our expectations of increasing time:




| Num Versions | Avg Time (ms) |
|---|---|
| 1 | 515.4 |
| 2 | 807.8 |
| 3 | 1143 |
| 4 | 1479 |
| 5 | 1871 |

**iv)** We observed the following results for measuring the time to insert the Wikipedia English Corpus into our distributed file system. We measured 5 data points with 4 machines and 5 data points with 8 machines, and calculated averages and standard deviations.




The averages and standard deviations are nearly equal. This is expected, as any put request to our file system goes through exactly 5 nodes (coordinator and 4 replicas), so having 4 machines or 8 machines in the system should have little effect on the overall performance of the put operation.