MP2 REPORT – GROUP 20 (gnarang2, hnehra2, sgangil2)

We implemented a simple distributed file system replicating a Hadoop system. Our system had a Name Node acting as the master which was responsible for storing all the files' metadata. Then there were data nodes which were responsible for storing the files and the replicas. We chose to store 3 replicas of each file along with storing the original file, and thus essentially each file had 4 copies in order to tolerate up to 3 simultaneous failures. Whenever a client put a "PUT" request first it would contact the master and get a list of locations where it would have to store the file. We made the client responsible for storing all the replicas of the file whenever a put operation was performed in order to guarantee presence of files. For GET operation, the client would contact the Name Node to get information regarding the location of the file after which it would fetch the file from the Data Node which the master provided. We did not use a majority of quorum system, instead as part of our file transfer protocol we sent and established a variable "updateCount" associated with each file which determined the latest file. This enabled us to use to provide the information about the latest file instantaneously at the client and further simplified replication which ran in the background observing the "updateCount" of each file at each node sending a replication request to whichever node which doesn't have the latest file. Furthermore, the replication was also responsible for sending replication requests to nodes whenever a node containing the file failed and the Name Node detecting that there were lesser number of files than were required to ensure 3 simultaneous failures. The Name Nodes' meta data structure also consisted of a "fileStatus" variable which could take three values "REPLICATING", "WRITING," and "READABLE" depicting that a file is replicating, or is being written, or is readable on a per node basis. The DataNode would send acknowledgements or updates to the Name Node indicating a change in UpdateCount or the Status at the end of every operation concerning the data node to ensure that the data in the Name Node is up to date with the data in the DataNode. For file transfer along with communication related to file transfer we used TCP messages and the failure detection worked through UDP messages as in MP1. The following figures represent how fast our PUT / GET operations were for the respective file sizes:

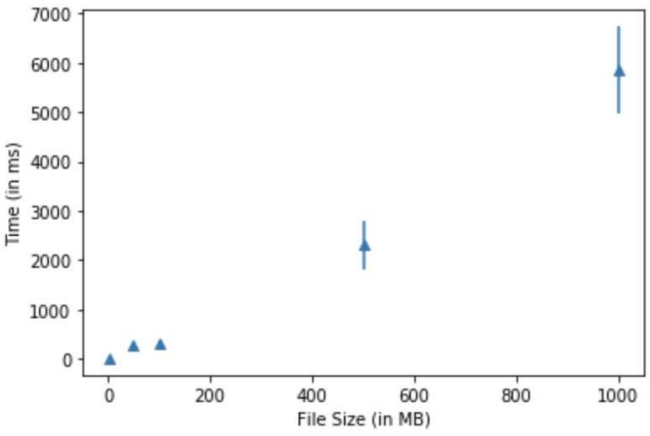


Figure 1: GET Operation

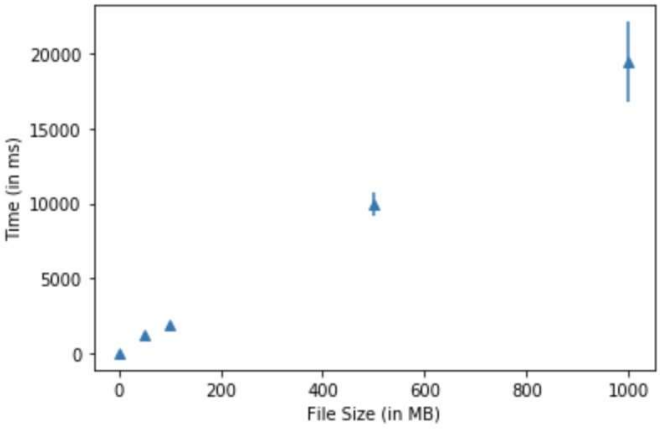
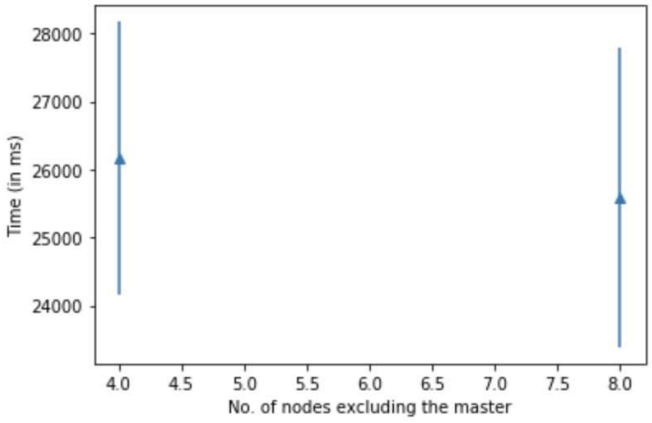


Figure 2: PUT operation

Although Standard Deviations cannot be seen for the X axis values of 1, 50, 100, they are present but are so small that they cannot be seen on the graph. For GET, they are 3, 23, and 42 ms and for PUT they are 11, 76, and 22 ms.

Note: Here the PUT operation's time measured represents how long it took for 4 copies of the file to be stored in our Simple Distributed File System, while on the other hand our GET operation included the time taken to fetch a particular file of a particular size from the respective data node, and is thus almost four times lesser time consuming than the PUT operation.

The trends seen above are the ones we expected, we can see that there is an almost linear increase in the time taken to PUT a file as the size of the file increased. This was solely because for all the different file sizes our algorithm took an almost similar time to begin file transfer and necessary communications related to file transfer. The actual file transfer involved a linearly increasing time depending on the size of the file because due to memory limitations and TCP communication limitations we could only send a certain number of bytes per iteration (we fixed that to be 5MB per iteration). Essentially, that means that the file transfer of a file of 5MB would take around x seconds and would happen in one "for" loops iteration and a file transfer of 5MB would happen in roughly (very close to) 2*x seconds and would happen in two "for" loop iterations. Furthermore, as the file size increased there was some extra time added for transferring files just because of our way of storing files which is explained at the end of this file. Overall, the trends in the graphs are roughly linear as we had expected them to be.



The Wikipedia corpus was of roughly 1.3GB and took roughly 25-26 seconds to transfer the entire 'tar' file. There is a slight difference between the values for 4 machines and 8 machines as depicted on the figure on the right. Theoretically, based on our replication and algorithm design, because we stored only 4 copies of each file, the transfer time should be the same for 4 machines and 8 machines just because the amount of work done to put the file is the same. The fact that the 8 machines file putting time is less is just due to random chance, and the nodes selected by random in one instance which decreased the overall average time required.

File Transfer: After every 5MB transferred, the file was stored rather than at the end or after in every N bytes read in order to save heap space used which would overflow in some cased if we waited till the end of the communication to store the file or take much more time if we wrote every time we received N bytes. 5MB provided us with a good threshold in the situation of 10 machines and 1 replication per data node at a time because then for a particular node at max there could be around 50MB*2 = 100MB (due to our design depending on copying the arrays at a particular line) which would be tolerated by Java's hashmaps.