Our design uses RPC in go as the backbone. The main function is running a thread which waits for command such as put, delete and get, once it gets the command, it will run the corresponding function to react to the command.

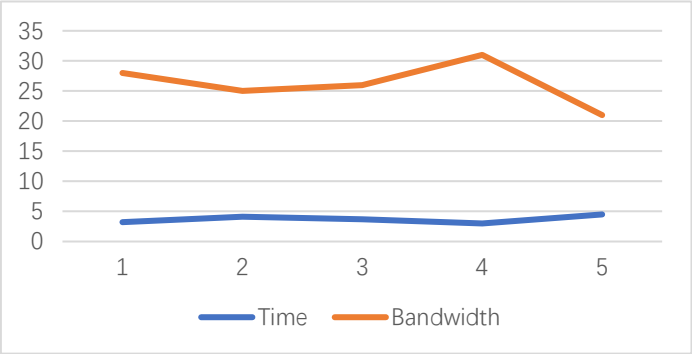
Our design has masters in the group which was elected by id and they are the same to others except that they contain the information including the host where a DFSFile exists. We can save memory because we save the file location only at the master not everywhere. Besides, our main function has used the MP2 to detect failure, once a master detects a failure, it will run the function of replication. It will first find out what file is on that machine, then for each file, it will find a next neighbor to replicate. Then the master will call a rpc function of a living machine and asks the living machine to send this file to the destination where the replication should be. After this, masters will update the file information to keep the system correct.

When a file is put, it will hash the filename to find where it should be replicated. And it will find two living neighbors to store this file, so this file will be stored at three different machines and it can tolerate up to 3 failures. When a delete/get command is entered, it will first find out where the file is stored, either by asking the master or find out by itself if it is master. Then it will directly ask the machine to delete/send back this file. We think this design can lessen the burden on the master because a machine will only ask the master for file information and this machine do the remaining operation itself. Upon actions which will change the file table, the corresponding machine will notify the master and master will receive an update about this information so the table will always be correct. When a ls command is entered, the slave machine will ask the master for the file table and print it out.

MP1 helps us debug because we can know the action taken by each machine when receiving a command and we can figure out whether it is correct.

1) Re-replication time and bandwidth

	1	2	3	4	5	Average	Dev
Time\s	3.2	4.1	3.7	3.0	4.5	3.7	0.62
Bandwidth\Bps	28M	25M	26M	31M	21M	26.2M	3.70

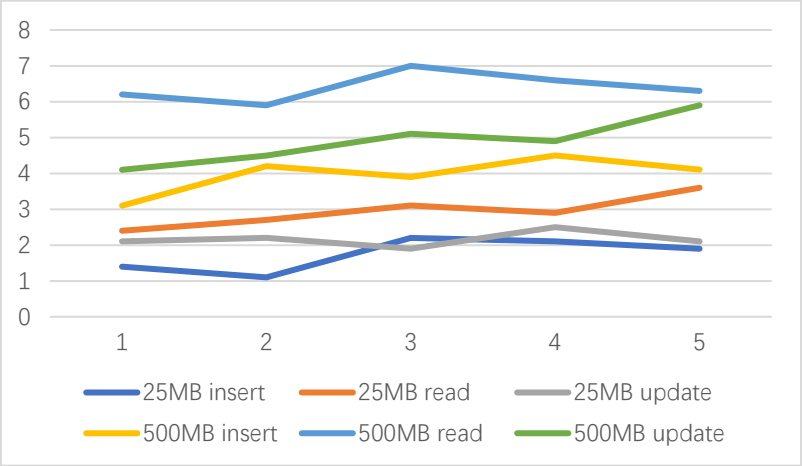


Re-replication time is a little bit more than 3 seconds which is close to the failure time out value we have set. It is because we are doing Re-replication after time out and the transmission time is relatively short, so we can get a value a little bit more than 3 seconds.

2) Times to insert, read, and update, file of size 25 MB, 500 MB

25MB	1	2	3	4	5	Average	Dev

Insert\s	1.4	1.1	2.2	2.1	1.9	1.74	0.472229
Read\s	2.4	2.7	3.1	2.9	3.6	2.94	0.450555
Update\s	2.1	2.2	1.9	2.5	2.1	2.16	0.219089
500MB	1	2	3	4	5		
Insert\s	3.1	4.2	3.9	4.5	4.1	3.96	0.527257
Read\s	6.2	5.9	7.0	6.6	6.3	6.4	0.41833
Update\s	4.1	4.5	5.1	4.9	5.9	4.9	0.678233

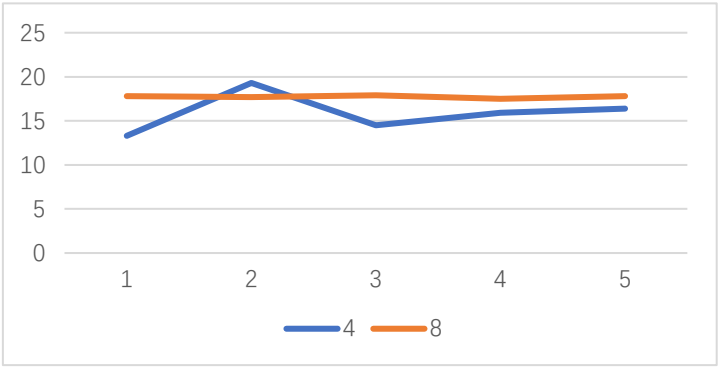


The time of transmit a 25MB file is short and will end before the link has reached its capacity so the bandwidth usage is not very high and is not very stable. However, for a much larger file like 500MB, bandwidth usage is much more stable and much higher.

The time to insert is similar to update, the reason is that in our design insert is pretty much the same as update. However, the read is slower than update, the reason may be that the bandwidth using for uploading and downloading is different.

3) Time to store the entire English Wikipedia corpus into SDFS with 4 machines and 8 machines

	1	2	3	4	5	Average	Dev
4	13.3	19.3	14.5	15.9	16.4	15.88	2.265392
8	17.8	17.7	17.9	17.5	17.8	17.74	0.151658



The time of transmit the Wiki file in a 4-machine system and 8-machine system is pretty close to each other because our design is irrelative to the size of the system and can handle different system size which shows the scalability.