CS425 / ECE 428: Distributed System, Fall 2022
MP 3 Simple Distributed File System

Name: Hang Yu (hangy6), Tian Luan (tian23)

Design Outline

We utilized Python to implement an HDFS-like Simple Distributed File System (SDFS), which is a flat file system. SDFS is easy to scale as the number of services increases, and it also contains consistency levels, failure tolerance, fast replication, and version control features. Each file (data) in SDFS is stored in four different machines to provide failure tolerance for up to three simultaneous failures. We also use consistency levels to support fast writing and reading. In the default settings, one write operation must be ack-ed by at least 3 replicas and one read operation need to be ack-ed by 2 replica, which means W = 3 and R = 2. This setting leads that two conflicting write operations will intersect in at least one replica, and one read and one write operation will also intersect in at least one replica, which guarantees that the user could always read the latest data. To implement the totally order update of data, we utilize a master node (which is called Namenode in SDFS). The master node can also be a regular node (which is called Datanode in SDFS). The client directly contacts Namenode when executing operations. When a Namenode fails, we will re-elect a new Namenode by using an algorithm like the bully algorithm, this will be executed by a failure detector, and the machine with the highest hostname will become the new Namenode (leader).

To be more specific, for the "put" operations, the client will contact Namenode to get a list of Datanode which need to store the file. And the Client will contact one of the Datanodes to send the file, the selected Datanode will be responsible to forward the received file to the next Datanode. In this way, SDFS can leverage a pipeline to transfer the files so we could get a good speed to store files. When a Datanode gets the file data, it will send an acknowledgment to Namenode, and when the Namenode receives at least three acks, it will tell the client the write is successful. For the "get" operations, the client will contact Namenode to get the list of replicas that contains the target file. And it will select two of them to get the data and compare them, then store the latest one locally. For the "delete" operation, the client will contact the Namenode and Namenode will be responsible for contacting the Datanodes which contain the files to delete all versions of this file. For "get-versions" operations, it is very similar to the "get" operation but the client will get more versions of the target file to store locally, and the local file name will be separated by delimiters. For other operations - "ls" and "store", the client will directly contact the Namenode to get the information.

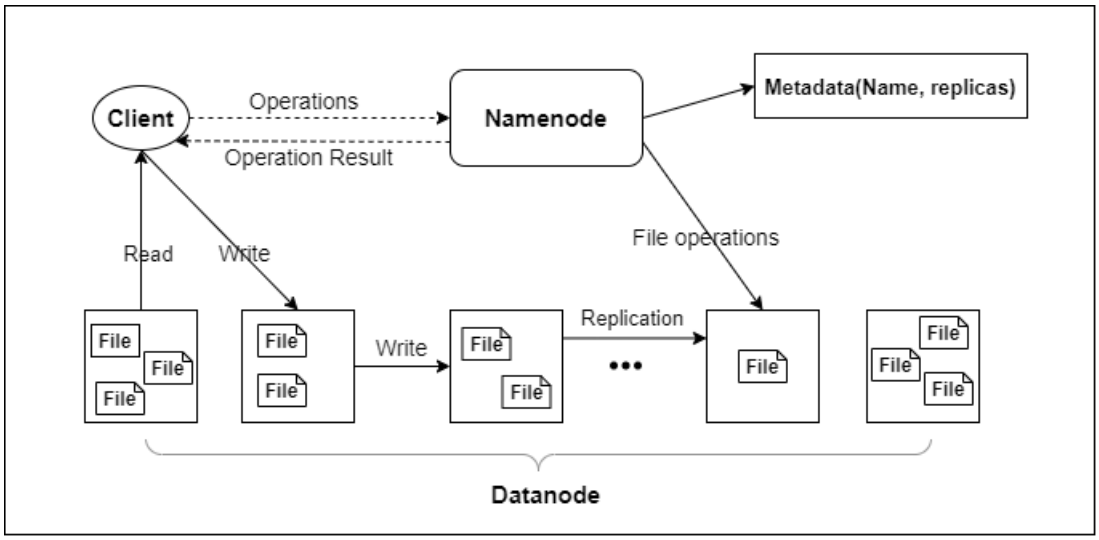

Figure 1: Design of Simple Distributed File System

Past MP Use

We followed exactly the same logic and reimplemented MP2 using Python to serve as a failure detector object. For our simple distributed file system, we want to know where to store the replicas so we pull the current membership list from the failure detector. The coordinator/master of our SDFS is the introducer of the failure detector. In this way, we can easily detect failures and maintain the topology information in our coordinator node.

For debugging, we used MP1 Python version to log the activity on each node just like print statement in a single process. Here, modified a little bit from MP1 and use grep on a single machine to collect how many errors, exceptions, etc are there on which VM. From the collected information, we can look into the details about the error log on that VM.

Measurements

i. re-replication time upon a failure of a 40MB file is 11.341 the average bandwidth is 386 MB/s during the replication. However, this is inaccurate since there are many UDP transmissions at the same time which is also counted in.

ii. times to insert, read, and update, file of size 25 MB, 500 MB (6 total data points), under no failure;

	insert	read	update
25 MB	1.635 ± 0.854	0.083 ± 0.042	2.676 ± 1.297
500 MB	30.054 ± 11.324	8.237 ± 5.637	15.524 ± 23.654

iii. Plot the time to perform get versions as a function of num-versions. The left figure is the time of put the files into the SDFS. The x-axis is the size of files to put measured in MB, the y-axis is the time to to put the files measured in second. We run experiments for each size of the data for five times and take the average and calculate the standard deviation. The blue line is the average time and the right line is the standard deviation. The right figure is the time to get n-versions of the file in SDFS vs. the number of versions N to retrieve. The standard deviation is represented by the shallow blue area. Also, we used a file with 40 MB size to repeat the experiment for five times for each datapoint and take the average and the standard deviation.

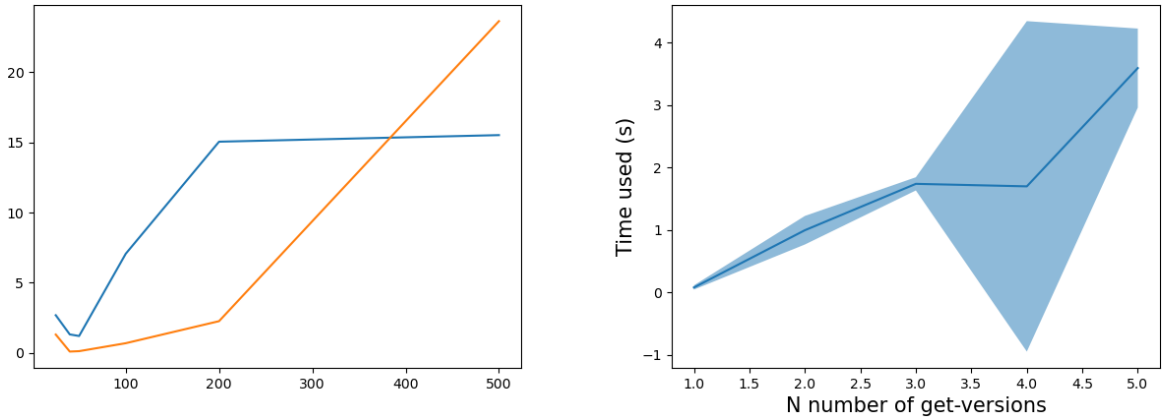

Figure 2: Figures of PUT file time vs file size (left) and figure of the time to get n recent versions (right)

iv. time to store the entire English Wikipedia corpus (1.3 GB) into SDFS with (not counting the master) 4 machines is 33.372 ± 18.534 s and with 8 machines is 36.249 ± 20.533 s

Analysis and Conclusion

From the results upon the four measurements, we found that in our SDFS, if the file size is small, then the time to put the file into the system is not stable.

i In the left figure of Fig 2, we can see that the average used time could be smaller for a larger file under 100 MB. This is counter intuitive, but it could also be subjected to some sudden variation in the VM network connection. The speed of network could changed during our experiments. **In general, the time to put the file into the distributed system is larger but not proportionally increase as the file size grows.** A more obvious trend shows that the standard deviation grows almost proportionally as the file size grows. This means, when we do the experiments, the speed of putting a larger file can vary a lot. We hypothesize that this is due to the variation in the network because it takes longer to send the files which amplify the impact of network changing.

ii In the right figure we can see that the average time of get N versions of a file in our SDFS is almost proportional to the number N. Similarly, we observe a larger variance in the retrieve time when N is larger which is also expected.

iii When we used 4 and 8 machines (not the master node) to store the entire English Wikipedia corpus, the time taken is longer than any experiments we have as expected and it is close to the linear interpolation of our other experiments data. Also, interestingly, we found that have 4 data node machines typically stores faster than that having 8 data node machines. We assume that this is caused by more communications between master node and different data nodes which occupies a little bit bandwidth resource given limited network.