MP3 Report

Linxi Li(linxili2), Gan Yao(ganyao2)

In the MP3, we designed and implemented a Simple Distributed File System(SDFS), which supports the following file operations: ***Put***, by which client is able to write local files into SDFS; ***Get***, by which client is able to retrieve files from SDFS to local disk; ***Delete***, by which client is able to delete files in SDFS.

Design

Our SDFS employs one-coordinator structure, meaning that it consists of one coordinator and many data nodes. Coordinator plays an important role in all of the operations including reading, writing, and delete. It decides which nodes that a given file should be stored on, maintains the meta data of SDFS that records locations of all files, and also handles re-replication when failures are detected. To tolerate up to 3 simultaneous failures, each file is replicated 3 times, making it total of 4 copies of each file stored in SDFS. To make write and read fast, consistency level of write and read are set to 3 and 2 respectively. That is to say, a write operation is considered done when it is ack-ed by 3 replicas; a read operation is considered done when it is ack-ed by 2 replicas.

When a client tries to write a new local file into SDFS, it first sends a write request to coordinator. Coordinator hash the file name to get 1 data node number, that data node's 3 successors are automatically chosen as replica nodes. And then the coordinator sends this list of 4 data node numbers back to the client. Client will directly contact the first data node in that list and transfer the file to-be-written to it. The first data node will be responsible for transferring this file to its three successors. Each of the 4 data nodes will send back an ack to coordinator when writing is done. Finally, coordinator will inform client that writing is done when it receives 3 acks, and a write operation is completed.

When a client tries to update an existing file in SDFS and sends request to coordinator, instead of hashing, coordinator directly get the list of data node numbers storing that file from meta data, and send the list to client. The rest of the operation is identical with writing a new file.

When a client tries to read an existing file in SDFS, it first sends a read request to coordinator. Coordinator fetches the list of data node numbers from meta data and sends back to client. Client then contacts any 2 of 4 data nodes in that list for their latest version number of the requested file. Client compare the 2 version numbers and ask the data node with larger version number for that file.

When a client tries to delete an existing file in SDFS, it sends delete request to coordinator. Coordinator then tells all of 4 data nodes storing that file to delete and wait for ack. When all 4 acks are received, coordinator reply to client and deleting is completed.

In case of the coordinators' failure, we implement an election mechanism based on Bully algorithm. When coordinator fails, nodes that detect that failure will start election. The current living node with the highest node number will be elected as the new coordinator. To make sure new coordinator function correctly, meta data is

periodically backed up on data nodes, so that new coordinator can ask data nodes for the latest meta data back-up. Whenever a client wants to contact coordinator, it needs to first contact election mechanism for current coordinators' number.

To make sure there are always 4 replicas of one file in SDFS, we implement a re-replication mechanism that is activated upon failure. Whenever coordinator is informed of failure of a data node, it will refer to meta data to find which files were stored on that failed data node. For each of those files, coordinator contact one other living data node possessing the same file, and instruct it to replicate that file to another node that does not possess the file. What about coordinator's failure? When coordinator fails, election mechanism will hold all failure messages until new coordinator is elected. Election mechanism then sends these failure messages to new coordinator and let it handle re-replications, including those files stored on old coordinator.

Past MP Use

MP2, membership service, plays an important role in the development of SDFS. Note that as mentioned above, all nodes are referred to as node numbers within SDFS, while IP address is needed for communications. So all nodes need to contact membership service for mapping node numbers to IP addresses. In addition, membership service is responsible for detecting all failures, and sending out all notifications of failures to coordinator and election mechanism. In a word, SDFS can never work correctly without membership service.

We also used MP1 to debug in later phase of our development, since it can help us track all the update events in all the log files distributed on each machine.

Measurements

(i) Re-replication time and bandwidth usage upon a failure, one 40 MB file.

	Average	Standard Deviation
Re-replication time(s)	3.68	0.116275535
Bandwidth usage(Bps)	17.83845	1.917821305

Above chart shows the average value and standard deviation among five trials, of re-replication time and bandwidth usage upon a failure of one node storing a 40MB file. We use linux command *time*, and linux software *iftop* to measure time and bandwidth respectively. Re-replication time is mostly determined by failure detection time out of membership service. Our membership service has failure detection timeout of 3 seconds, so the replication time slightly higher. Bandwidth usage is 2-second average after a failure detected by coordinator. The bandwidth usage is mostly determined by the size of file that needs to be re-replicated. When files are large enough, bandwidth usage caused by other messages are negligible.

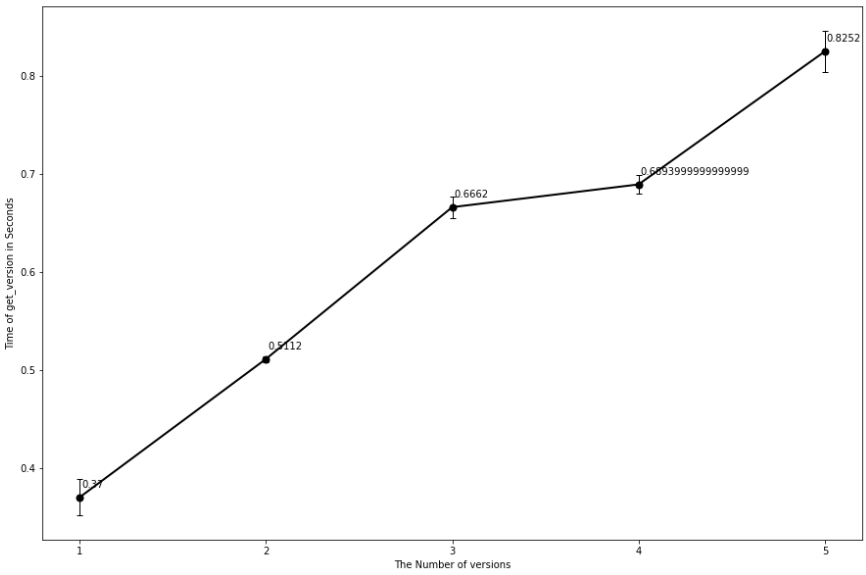
(ii) Time to insert, read, and update file of 25MB and 500MB.

Time(s)	Insert		Read		Update	
	25MB	500MB	25MB	500MB	25MB	500MB
Average	0.5964	7.5118	0.3598	19.2978	0.3778	6.8176
Std Dev	0.08081	0.78122	0.01406	10.8056	0.01594	1.13295

Above chart shows the average value and standard deviation among five trials, of

inserting, reading and updating file of size 25MB and 500MB. Generally speaking, operations on large files take more time than that on small files, which make sense. And update takes less time than insert. This may be because when writing new files, it takes coordinator some time to assign new nodes to that file. However, for read operation, it's slower than insert for 500MB file, but faster than insert for 25MB file. And the variation of measurements on 500MB file read is relatively large. We are not sure about the reason behind this. But it maybe be cause the difference in upload bandwidth and download bandwidth.

(iii)Time to perform get-version



Above plot shows the execution time of get-version with different num-version values. The time of get-version monotonically increases as the num-version increases. This trend is within our expectations, since as the number of versions increases, the SDFS needs to retrieve and write more files.

(iv)Time to store English Wikipedia corpus into SDFS with 4 machines and 8 machines

Time(s)	4 Machine	8 Machine
Average	55.9956	56.4764
Standard Deviation	2.809187	0.858491

To insert the whole Wikipedia Corpus into the SDFS, we used the **write** function of our SDFS, the time of inserting was surprisingly faster than we expected, both 4 Machines and 8 machines implemented the inset within 60 seconds. And their time are about the same. This meets our expectation since the volume of data that needs to be transferred is independent of number of machines. However, 8 machines showed more stabilized time (less standard deviation) than 4 machines. This maybe because when there are more machines, the transferring speed is less affected by the unstable network condition of several machines.