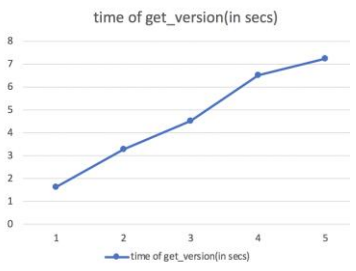
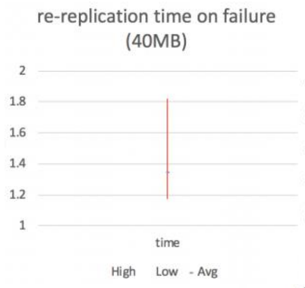
Design:
Our SDFS has 4 major components: udp-based group membership and failure detection algorithms, tcp-based remote procedure call for distributed grep and reliable message transfer, SDFS manager to handle distributed file system logic, and a CLIent component to interact with user.
Failure Detection and Group Membership: Same as completed in MP2.
Leader Election:
Our leader election algorithm is choosing lowest machine number based on the group membership information. Once a current leader's failure is detected, all remaining processes will vote for the lowest machine number they know. Once a node receive votes, it will use quorum to decide whether it is elected as the new lead. If true, it will send messages to all nodes asking them to update their master. Thus the next lowest machine number remaining in the membership list will be elected as new master/leader.
Distributed FileSystem Logic:
We support the following commands: PUT, GET, DELETE, LS, and STORE, GET-VERSIONS
PUT path/localfilename sdfsfilename : this request first is routed to master. Master checks that if the target sdfs file was updated in 60 seconds before. If true, then the master would ask the client for update confirmation, yes or no. If yes, the master will update the 4 replicas. If not updated in 60 secs, the master will select 4 different target nodes to store this file and then return this information to the client and store the metadata. If the local file was put before, then the master return the replica address as reply information. Once client received this information, it will send the file data to those 4 target nodes. When target node received the data, it will save the data to its own storage and record its metadata. At client side, it will use QUORUM to decide when the PUT is finished, that is if we put to 4 replicas, we return success once 3 are done.
GET sdfsfilename localfilename: first ask master to get who has the file and contact them to get file data. QUORUM is also used here, that is if we read from 4 replicas, we return the one with highest version once 3 copies are received. During this process, if some replicas version is less than the latest version, we'll also initiate a stale version repair for that node.
DELETE sdfsfilename: first ask master to get a list of who has the file. When master get this request, the master will remove this file's entry from the metadata. The client will then ask the replicas in this list to delete the file and remove from its local metadata.
LS sdfsfilename: first ask master to get a list of who has the file. When master get this request, it will return client the list which contains the replica file's address. Then client just list it to screen.
STORE: check SDFS file list on the current process and list them on screen.
GET-VERSIONS sdfsfilename num-versions localfilename: Since the requirement says that the version needed to fetch is no more than 5, we store versions into five cache files, CACHE0, CACHE1, CACHE2, CACHE3, CACHE4. For each version v, we store it into the CACHE file with id (v-1)%5. When fetching, we start from the largest version and decrease 1 per time by module 5. For example, if the largest version id is 7, then we would start fetching from CACHE with id (7-1)%5 = 1, and then 0, then 4,3,2. Finally we got the latest 5 versions and merge them into one local file.
Replication Strategy: Our SDFS is tolerant up to 3 machine failures at at a time, therefore, we want to maintain 4 replica per file. After failure(s) get detected or stale version detected, we initialize a repair process to copy data from the remaining node to new nodes that are still alive. Therefore, at all time we try to keep 4 replica per file.
Usefulness of MP1:MP1 is useful for us. It provides us a great convenience to get local log messages without having to log into the machine and type in multiple commands to retrieve and grep the log.

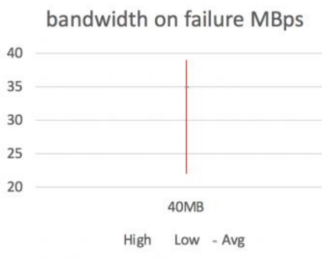
Measurements: When plotting time to insert, read, update, the time for each round is average of 6 data points.

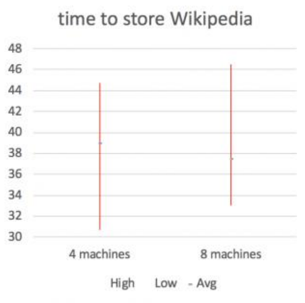
Rounds	1	2	3	4	5	avg	std
re-replication time on failure (40MB)	1.18	1.27	1.82	1.17	1.35	1.358	0.240
bandwidth on failure MBps	39.3	35.9	22.8	37.9	33.8	35.5	6.043
time to insert 25MB	1.84	1.91	2.99	1.82	1.83	2.078	0.457
Time to insert 500MB	16.9	22.5	26.2	31.6	26.9	24.8	4.910
time to read 25MB	1.85	1.68	1.67	1.72	1.71	1.72	0.067
time to read 500MB	7.13	7.95	7.04	7.58	7.12	7.35	0.354
time to update 25MB	1.85	1.88	2.27	2.82	1.91	2.12	0.463
time to update 500MB	25.7	27.8	22.8	30.4	35.6	29.6	5.732
time to detect write-write conflict	1.03E-05	1.09E-05	9.77E-06	1.02E-05	9.26E-06	1.01E-05	5.51E-07
time to store Wikipedia with 4 machines	43.04	44.77	44.78	39.71	30.65	40.58	5.303
time to store Wikipedia with 8 machines	37.89	46.75	30.23	31.64	40.62	36.98	6.134

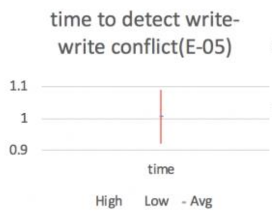
Plot the time of get-versions as a function of num-versions. We use 25MB to test.

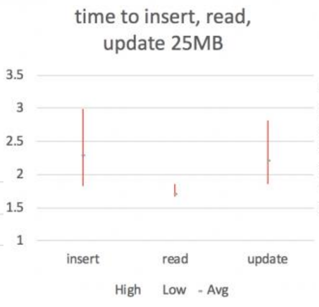


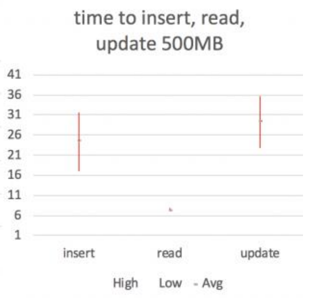












Discussion

(1) Bandwidth is slightly less than 40MBps and the time cost is slightly more than 1s, which is expected since the total amount of traffic is 40MB.

(2) Time to insert and update is close, which is as expected since the underlying mechanism is the same. Time to read is typically less than time to insert, this may be the result that we only need to actually transfer 1 copy of file to the client instead of 2 (when inserting).

(3) Time to detect write conflict is trivial. This is because the master can check the record quickly, without reading the file on disk.

(4) Time of get version function is nearly linear with the increase of version number. When only one version is required, the average time on one version is relatively high, the reason might be the file initialization takes extra time.

(5) Time to store the Wikipedia corpus: there is no significant difference between a 4-machine system and an 8-machine system. The main time cost comes from file transfer through network and there is always 3 replica needed to be transfer regardless the number of machines.